

A Smarter Stupid Computer

Alexander Wasey



MInf Project (Part 2) Report

Master of Informatics
School of Informatics
University of Edinburgh

2022

Abstract

This MInf part 2 project report covers my progress in enhancing the “stupid computer” program. It is based on a technique taught to students taking Informatics 1-A at the University of Edinburgh, which seeks to help them understand the structure and execution of Haskell functions. Given the expression `map square [1,2,3]` the program outputs the following trace:

```
map square [1, 2, 3]
= (square 1) : (map square [2, 3])
= (1 * 1) : (map square [2, 3])
= 1 : (map square [2, 3])
= 1 : ((square 2) : (map square [3]))
= 1 : ((2 * 2) : (map square [3]))
= 1 : (4 : (map square [3]))
= 1 : (4 : ((square 3) : (map square [])))
= 1 : (4 : ((3 * 3) : (map square [])))
= 1 : (4 : (9 : (map square [])))
= 1 : (4 : (9 : []))
= 1 : (4 : ([9]))
= 1 : ([4, 9])
= [1, 4, 9]
```

This report covers adding support for lazy evaluation to the program, which included support for `let` statements. Support has also been added for case statements and lambda abstractions. Users can now define custom infix operators and their own data types. These features required a new formal-actual mapping system, and improvements to GHC interpretation. The program was then tested using 46 code examples, which showed a large improvement over the original implementation. Feedback was sought from the target user base, by issuing a survey to first-year students, which received 37 respondents. However the program is lacking support for `where` statements, and has issues with operator precedence.

Acknowledgements

I'd like to thank my supervisor Philip Wadler for his feedback, guidance and support. I would also like to thank Dylan Thinnes for his advice throughout.

Contents

1	Introduction	1
1.1	The Stupid Computer	1
1.2	Summary of results	2
1.3	Report summary	2
2	Previous work	3
2.1	Design	3
2.1.1	Overview	3
2.1.2	Reduction strategy	3
2.1.3	How reductions work	4
2.2	Implementation	7
2.2.1	Overview of approach	7
2.2.2	Parsing	8
2.2.3	Input validity checking	9
2.2.4	Reducing function applications	9
2.2.5	GHC interpretation	10
3	Improvements to the stupid computer	11
3.1	Lazy evaluation	11
3.1.1	call-by-name	11
3.1.2	Sharing	14
3.1.3	Implementation	14
3.2	Improved user interface	16
3.3	New formal-actual mapping system	18
3.4	Reparsing of GHC interpretation results	21
3.5	Improved feature support	21
3.5.1	Let statements	21
3.5.2	Lambda abstractions	24
3.5.3	Case statement support	25
3.5.4	Custom infix operator support	27
3.5.5	Support for user defined data types	27
3.6	Current system limitations	28
3.6.1	Failure to evaluate guard values	28
3.6.2	Lack of support for where statements	29
3.6.3	Incorrect operator precedence	30

4	Evaluation	32
4.1	Testing	32
4.2	User Feedback	32
4.2.1	Part one	33
4.2.2	Part two	33
5	Conclusions	35
	Bibliography	36
A	Additional survey documents	38
A.1	Stupid Computer Overview	39
A.2	Installation guide	40
A.3	Participant Information Sheet	44

Chapter 1

Introduction

1.1 The Stupid Computer

In the first half of this project I developed a program called the stupid computer, which produces traces of Haskell programs [Wasey, 2021]. (This report assumes some familiarity with Haskell, if not the book “Learn You a Haskell” [Lipovaca, 2011] is an excellent resource.) It is designed to help Informatics 1A students better understand the behaviour of Haskell functions. This course introduces first year Informatics students to programming at the University of Edinburgh, with around 450 students taking the class each year [Wadler et al., 2022]. For example if a student is learning about the map function they may first look at its definition:

```
map :: (a -> b) -> [a] -> [b]
map f (x:xs) = f x : map f xs
map f [] = []
```

For a student with limited programming experience, this definition may not be sufficient for an understanding of how the function behaves in practice. The stupid computer helps by producing an execution trace for the function, such as the following:

```
map square [1, 2, 3]
= (square 1) : (map square [2, 3])
= (1 * 1) : (map square [2, 3])
= 1 : (map square [2, 3])
= 1 : ((square 2) : (map square [3]))
= 1 : ((2 * 2) : (map square [3]))
= 1 : (4 : (map square [3]))
= 1 : (4 : ((square 3) : (map square [])))
= 1 : (4 : ((3 * 3) : (map square [])))
= 1 : (4 : (9 : (map square [])))
= 1 : (4 : (9 : []))
= 1 : (4 : ([9]))
= 1 : ([4, 9])
= [1, 4, 9]
```

The stupid computer is based upon an identically named technique used by students in Informatics 1-A. This course teaches Haskell to first year students, and the technique is used widely both within the course and its textbook “Introduction to Computation” [Sannella et al., 2022]. The overall aim for the project is to produce a program which can automate this technique, a basic implementation of this was achieved in the first year of the project. The goal for this year of the project is to further enhance the stupid computer, primarily by adding support for lazy evaluation [Friedman et al., 1976][Henderson and Morris Jr, 1976].

1.2 Summary of results

The stupid computer has been extended to include support for lazy evaluation. This has necessitated the implementation of a new formal-actual mapping system. The new system is capable of mapping actuals which are not fully reduced, overcoming a limitation from the first year of the project. This is implemented by comparing the ASTs of arguments and patterns.

Adding lazy evaluation has also necessitated supporting let statements. The semantics of these has been based on the call-by-need lambda calculus, with modifications for a Haskell context [Maraist et al., 1998].

In addition to this support has been added for case statements, lambda abstractions and custom infix operators. These have been implemented using the functionality developed for the implementation of lazy evaluation. There is also improved support for using functions from the Haskell prelude. Results from these functions can now be reused, as they are reparsed by GHC. Users can now make use of custom data types, which has been enabled by improvements to GHC evaluation. A brand new user interface has been added, which now allows users to use standard Haskell source files.

However there is still a lack of support for where statements and issues with operator precedence.

1.3 Report summary

Chapter 2 provides an overview of the stupid computer, as it was implemented in the first year of the project. Following this chapter 3 contains the improvements made to the stupid computer, and details of their implementation. This includes support for lazy evaluation, let statements, case statements, and lambda abstractions. These are supported by a new formal-actual mapping system, and improvements to GHC interpretation. This chapter also includes an overview of the system’s limitations. Chapter 4 describes how I evaluated the stupid computer. This included both written tests, and user feedback. Chapter 5 concludes with a number of possible future improvements to the system.

Chapter 2

Previous work

This chapter gives a brief overview of the design and implementation of the stupid computer, as achieved in the first year of the project [Wasey, 2021].

2.1 Design

2.1.1 Overview

To use the original stupid computer the user would first provide it with a number of function definitions, such as this definition for `sum`.

```
sum :: Num a => [a] -> a
sum (x:xs) = x + sum xs
sum [] = 0
```

The user would also provide a single Haskell expression, such as:

```
sum [1,2]
```

The program would then output a series of Haskell expressions:

```
= sum [1,2]
= 1 + sum [2]
= 1 + 2 + sum []
= 1 + 2 + 0
= 1 + 2
= 3
```

The first line is the expression the user originally provided, each following line represents a single reduction step. The trace finishes when no more reductions are possible.

2.1.2 Reduction strategy

The original stupid computer followed a mostly call-by-value reduction strategy, which reduces the leftmost innermost redex at each step. This means that a function's argu-

ments are always reduced before the function itself. For example given the multiply function:

```
multiply x y = x * y
```

Then the reduction of `multiply (1+2) (3+4)` is:

```
multiply (1+2) (3+4)
= multiply 3 (3+4)
= multiply 3 7
= 3 * 7
= 21
```

2.1.3 How reductions work

This section gives an overview of how different types of expressions were reduced in the original implementation. These approaches were designed with the following goals in mind:

1. At each step of the reduction the output should be valid Haskell such that the users could, if they wished, execute it with the Haskell compiler (GHC).
2. Identifiers should be kept the same, such that users can easily relate the output to their own code.
3. The structure of the users code should be represented in the output, for example don't change an if-then-else statement to the guarded equivalent.
4. Try and only show reductions which are informative to the user. For example if `sum` is defined as:

```
sum xs = if (length xs >= 1)
         then (head xs) + sum (tail xs)
         else 0
```

In this case the user is unlikely to want to view the full reductions of `length`, especially as for a list of length n it will take $2n + 1$ steps!

2.1.3.1 User defined functions

Function applications using user defined functions are expanded. The function application is replaced with the function body, where the formal arguments have been replaced by the actual arguments from the application. For example if the user evaluates `multiply 5 4` using the previous definition of `multiply` the first reduction step will be:

```
multiply 5 4
= 5 * 4
```

Pattern matching ([Jones, 2003], Chapter 3.17) slightly complicates this. It requires a more complex mapping between formal variables and actual parameters, and means

functions can have more than one body. Using the definition of `sum` as an example, in the first body the formal `x` is mapped to the first item in the input list, and `xs` to the rest of the items in that list.

```
sum :: Num a => [a] -> a
sum (x:xs) = x + sum xs
sum [] = 0
```

When pattern matching is encountered, the original stupid computer simply expands to the correct body given the arguments. For example if `sum` is applied to the list `[1, 2, 3]` the first body is expanded to:

```
sum [1,2,3]
= 1 + sum [2,3]
```

And if called with the empty list then the second body is expanded to:

```
sum []
= 0
```

Guards ([Jones, 2003], Chapter 4.4.3) are another way functions can have multiple bodies. For example the `factorial` function can be defined with guards:

```
factorial :: Int -> Int
factorial n | n >= 2 = n * factorial (n-1)
            | otherwise = 1
```

The first guard condition which is satisfied has its body selected, note that an `otherwise` condition is treated as always being true. This series of evaluations are not shown to the user, to do so would have broken the previously mentioned design goals. I found that all the methods I considered would have either not been valid Haskell, or changed the structure of the users code significantly. Therefore the stupid computer simply expands directly to the correct body, for example with `factorial 5`:

```
factorial 5
= 5 * factorial (5-1)
```

However not showing the evaluation of the guard, even when the guard is complex and requires many reduction steps, is a large disadvantage. In this case I believe it is better to stick to the design goals, even if it results in reductions not being shown.

2.1.3.2 Prelude functions

When a function is not defined but the user, but by the Haskell standard library, the application is reduced directly to its result. In Haskell the standard library is known as the prelude [The GHC Team, 2001]. For example using the prelude function `product` gives the following reduction step:

```
product [1,2,3,4,5]
= 120
```

The result of the function application is acquired with GHC interpretation, as described in section 2.2.5.

It was decided to not produce full reductions for prelude functions as it could result in very unwieldy traces. For example a full reduction of `length` requires $2n + 1$ reduction steps, a user is unlikely to want to see this detail. In addition to this prelude functions can be defined in unexpected ways, such as this definition of `sum` from the prelude [The GHC Team, 2001].

```
sum :: Num a => t a -> a
sum = getSum #. foldMap Sum
```

This is because they are designed for performance and flexibility, not to be easily understood. However this means using these definitions could be very confusing for users.

2.1.3.3 If-then-else

In the case of if-then-else statements ([Jones, 2003], Chapter 3.6) only the condition is evaluated before the if statement. This avoids unnecessary evaluation, given that one of the two possible results will always be discarded. Such an example follows:

```
if (1 >= 2) then 1 * factorial (1-1) else 1
= if False then 1 * factorial (1-1) else 1
= 1
```

2.1.3.4 List comprehensions

List comprehensions [Turner, 2016] have three components: generators, guards, and a body. For example in the comprehension `[x*x | x <- [1,2], x > 1]` the body is `x*x`, which produces each element in the resulting list. `x <- [1,2]`, is a generator which feeds each value from its list into the body and guards. Multiple generators can be used, in which case all combinations of values from their lists are fed into the body. Finally `x > 1` is a guard, which must be satisfied for the body to be kept.

At each reduction step, the left most generator or condition is evaluated. Generators have their first value substituted into a copy of the comprehension. Guards are evaluated, if they are true the comprehension is kept, else it is discarded. So a full reduction of the comprehension is as follows:

```
= [x*x | x <- [1,2], x > 1]
= [1*1 | 1 > 1] ++ [x*x | x <- [2], x > 1]
= [1*1 | False] ++ [x*x | x <- [2], x > 1]
= [] ++ [x*x | x <- [2], x > 1]
= [] ++ [2*2 | 2 > 1 ]
= [] ++ [2*2 | True]
= [] ++ [2*2]
= [] ++ [4]
= [4]
```

2.1.3.5 Undefined functionality

When an expression has no defined reduction it is immediately reduced to its result. In the following an unsupported lambda abstraction is reduced:

```
(\x -> x * x) 3
= 9
```

This gives users flexibility to use as-yet unsupported functionality, this is enabled by evaluating the expression with GHC, see section 2.2.5.

2.2 Implementation

The stupid computer was implemented in Haskell, as it gives access to many tools for working with Haskell source code.

2.2.1 Overview of approach

The approach is based on manipulating abstract syntax trees, the user's input is parsed into an AST, and each reduction step is a manipulation of this tree. Below are the first three steps of reducing `sum [1, 2, 3, 4]` and the corresponding series of ASTs (Figure 2.1).

```
= sum [1, 2, 3, 4]
= 1 + sum [2, 3, 4]
= 1 + 2 + sum [3, 4]
```

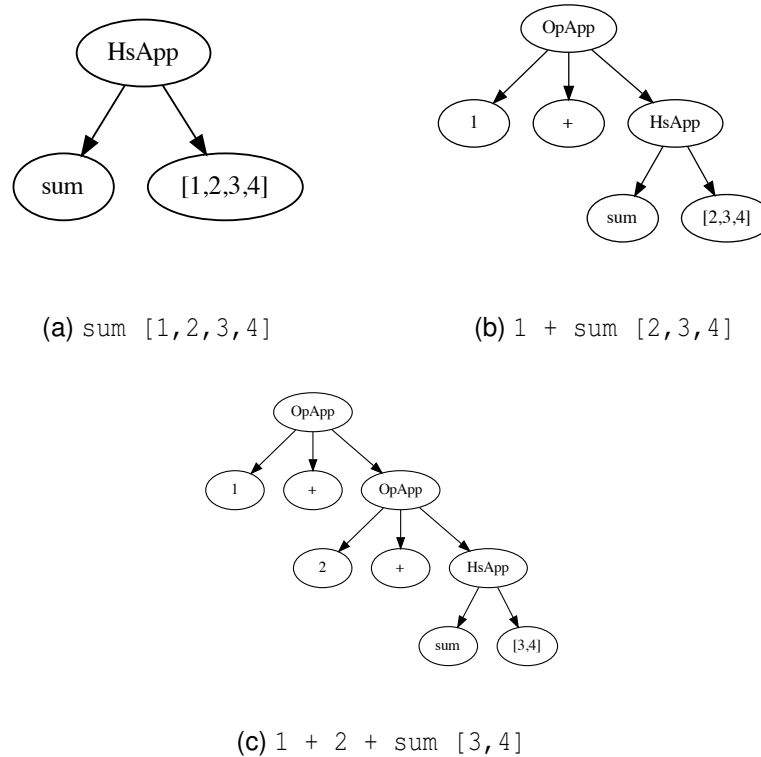


Figure 2.1: Series of (simplified) ASTs, each representing a reduction step.

At each reduction step the following occurs.

1. The appropriate redex is identified for reduction.
2. Its AST of its reduced form is constructed.
3. The reduced form is substituted into the AST in place of the redex.
4. The new AST is to converted back to readable Haskell source code by the pretty-printer. This is then shown to the user.
5. If no more reductions are possible then the reduction is finished, else return to step one.

2.2.2 Parsing

Before the reduction can begin the users input must be parsed into an AST. This is done by GHC, which provides an API which gives access to its lexer and parser [The GHC Team, 2002]. This avoided the development time associated with implementing my own version. The API also gives access to a pretty-printer to convert the AST to readable Haskell source code.

The main drawback of using the GHC AST is its relative complexity compared to the semantics of Haskell. For example in Haskell lists and strings are semantically identical, strings are just lists which contain characters. However in the GHC AST (as

generated by the parser) lists are represented by `ExplicitList` nodes, and strings by a `HsLit` nodes. These extra cases were not a large issue in the first year of the project, as I made heavy use of GHC interpretation. This meant I did not have to account for these extra cases in my code, as it was handled by GHC. In this year of the project this has become more of an issue, mainly when implementing the new formal-actual mapping system in section 3.3.

2.2.3 Input validity checking

Before any reductions occur the user's input is checked to ensure that it is valid Haskell. The parsing stage (section 2.2.2) checks that the input follows Haskell syntax, however it is also useful to know that the users code is semantically valid. To do this the users input expression is run with GHC interpretation. If this fails then the user's code is not semantically valid, so the stupid computer exits, with a message telling the user that their code is invalid.

2.2.4 Reducing function applications

The main changes made to the stupid computer this year involve the reduction of function applications, therefore this overview of their implementation is included for context.

1. The function arguments are first identified, with a depth first search of the function application subtree.
2. If any of the function arguments are not fully reduced, then they undergo reduction until they are.
3. The function body to be substituted into the main tree is identified. As seen previously this is complicated by pattern matching and guards. This is achieved by creating and evaluating a new function, for example the `gt` function has three possible bodies:

```
gt x [] = [] -- Body #0
gt x (y:ys) | y > x = y : gt x ys -- Body #1
             | otherwise = gt x ys -- Body #2
```

A new function is then created which returns the index of the appropriate body:

```
gt x [] = 0
gt x (y:ys) | y > x = 1
             | otherwise = 2
```

This can be evaluated by GHC (section 2.2.5) and the result mapped back to the original body.

4. A mapping must be created between the function's formal arguments and its actuals. For example if the function application is `gt 5 [1,2]` then the formal

actual map is: $x = 5$, $y = 1$, $ys = [2]$. This mapping is also acquired by creating a new function, for `gt` this is:

```
gt x [] = [("x", show x)]
gt x (y:ys) = [("x", show x), ("y", show y), ("ys", show ys)]
```

This function is evaluated with GHC and the result used as the formal-actual map.

5. The formals in the function body are substituted for the appropriate actual values.
6. The newly created subtree is substituted into the main AST in place of the function application.

2.2.5 GHC interpretation

As previously mentioned in section 2.1.3.5 the stupid computer can use GHC to evaluate expressions directly. This is achieved by the hint library [The Hint Authors, 2007], which uses the user's installation of GHC to interpret Haskell expressions at run time. The main limitation of this library is that the interpreter can only take expressions as input, not definitions. In the initial version of the program I worked around this limitation by placing user defined functions within a `let` expression. For example if evaluating `gt 2 [1,2,3]`, using the definition of `gt` seen previously, then the expression to be evaluated is:

```
let {gt x [] = []; gt x (y:ys) | y > x = y : gt x ys ;
    | otherwise = gt x ys} in gt 2 [1,2,3]
```

This approach was limited, and did not allow for custom data types, therefore a new approach has been taken in section 3.5.5.

Chapter 3

Improvements to the stupid computer

This chapter details the main changes that have been made to the stupid computer. It covers both the changes in behaviour, and the implementation changes required to enable them.

3.1 Lazy evaluation

Previously the stupid computer used a call-by-value strategy, however Haskell uses lazy evaluation [Friedman et al., 1976; Henderson and Morris Jr, 1976]. This is treated as a call-by-name strategy, with the addition of sharing. This allowed the design and implementation of each element to be completed separately. The semantics for lazy evaluation have been heavily based on the call-by-need lambda calculus, though adapted for a Haskell context [Maraist et al., 1998].

3.1.1 call-by-name

Previously the stupid computer used call-by-value evaluation, where the arguments to a function are reduced before the function application itself. For example the reduction of `multiply (2+3) (4+5)` in call-by-value is:

```
multiply (2+3) (4+5)
= multiply 5 (4+5)
= multiply 5 9
= 5 * 9
= 45
```

Whereas with call-by-name evaluation function applications are evaluated before their arguments, delaying evaluation of values until they are needed. The previous reduction using call-by-name will be:

```
multiply (2+3) (4+5)
= (2+3) * (4+5)
= 5 * (4+5)
= 5 * 9
```



```
= 45
```

While this is now the main evaluation order for the updated stupid computer, there are several cases where function arguments must still be reduced before the function itself.

3.1.1.1 Pattern match ambiguity

One situation is where it is unclear from the function arguments which function body should be selected. Using the `isZero` function as an example:

```
isZero 0 = True
isZero _ = False
```

If this is called as `isZero (1+2)` it is not possible to immediately determine which definition it should be expanded to. This is because the result of `1+2` isn't yet known, and as such it could match with either `_` or `0`. Therefore the argument must be evaluated until this ambiguity is removed.

Any ambiguities are detected by examining the definitions previous to the definition that will be expanded to. Which definition will be expanded to is determined by the system described in point 3 in section 2.2.4. if the pattern belonging to any of these definitions have the potential to match against the arguments then there is an ambiguity. In this example `isZero _` is the definition to be expanded to, but `isZero 0` has the potential to match against `1+2`. The pattern matching system described in section 3.3 can determine if an argument and pattern have the potential to match.

In this case only one reduction is required and therefore the full reduction is:

```
isZero (1+2)
= isZero 3
= False
```

3.1.1.2 Formal-actual mapping failure

Even when there is no ambiguity an argument may still need to be evaluated further. For example the `head` function only has one definition:

```
head (x:_) = x
```

However if called with an argument which doesn't directly match against the pattern `(x:_)`, such as `map square [1,2,3]`, then the function cannot be immediately reduced. Instead the argument must be reduced until it does match against the pattern. Whether or not the argument matches against the pattern is checked by the formal-actual mapping system, described later in section 3.3. Once the argument has been reduced enough such that it matches against the pattern the reduction can continue. In this case only one reduction is required, to `((square 1):(map square [2..]))`. At this point `x` can be mapped to `(square 1)`, as such the full reduction is:

```
head (map square [1..])
= head ((square 1):(map square [2..]))
= square 1
```

```
= 1 * 1
= 1
```

3.1.1.3 Prelude defined functions

While the call-by-name strategy is followed for all functions which are user defined, functions defined in the prelude still follow the call-by-value strategy. For example, if the user had defined `map` and `square`, but used the definition of `sum` from the standard prelude, then the reduction of `sum (map square [1,2])` is:

```
sum (map square [1, 2])
= sum ((square 1) : (map square [2]))
= sum ((1 * 1) : (map square [2]))
= sum (1 : (map square [2]))
= sum (1 : ((square 2) : (map square [])))
= sum (1 : ((2 * 2) : (map square [])))
= sum (1 : (4 : (map square [])))
= sum (1 : (4 : []))
= sum (1 : ([4]))
= sum ([1, 4])
= 5
```

This can be contrasted to the reduction when `sum` has been defined by the user:

```
sum (map square [1, 2])
= sum ((square 1) : (map square [2]))
= (square 1) + sum (map square [2])
= (1 * 1) + sum (map square [2])
= 1 + sum (map square [2])
= 1 + sum ((square 2) : (map square []))
= 1 + (square 2) + sum (map square [])
= 1 + (2 * 2) + sum (map square [])
= 1 + 4 + sum (map square [])
= 1 + 4 + sum []
= 1 + 4 + 0
= 1 + 4
= 5
```

Call-by-value is used for prelude defined functions because otherwise the user would not be able to see the reduction of the function arguments. This is because prelude defined functions are not expanded, but instead reduced directly to their result, for reasons outlined in section 2.1.3.2. Call-by-name would result in the following, where the reduction of `map square [1,2]` is omitted: (In this case `sum` is defined in the prelude.)

```
sum (map square [1,2])
= 5
```

3.1.2 Sharing

Call-by-name alone is not efficient, and can result in the repeated computation of values, which wasn't an issue with call-by-value. For example if reducing the expression `square (2 * 3)` with call-by-name:

```
square (2 * 3)
= (2 * 3) * (2 * 3)
= 6 * (2 * 3)
= 6 * 6
= 36
```

Then the calculation of `(2 * 3)` is been repeated twice, which is not the case in call-by-value. This can be addressed by adding sharing (giving us lazy evaluation). Here the calculation of `(2 * 3)` will be placed within a `let` statement, allowing it to only be calculated once. The full reduction using lazy evaluation is:

```
square (2 * 3)
= let x_0 = (2 * 3) in x_0 * x_0
= let x_0 = 6 in x_0 * x_0
= let x_0 = 6 in 6 * x_0
= let x_0 = 6 in 6 * 6
= 6 * 6
= 36
```

These `let` statements are only inserted when the following conditions are met.

1. The actual argument is not fully reduced.
2. The formal argument appears multiple times within the function definition.

The call-by-need lambda calculus [Maraist et al., 1998] does not have these conditions, introducing `let` statements each time a function is reduced. However adding these conditions reduces the number of unnecessary lets introduced, which helps keep the traces as simple as possible.

3.1.3 Implementation

Implementing lazy evaluation required major changes to the reduction process for user defined functions (see section 2.2.4).

3.1.3.1 Implementing call-by-name

Call-by-name was implemented by replacing step 2 of function application reduction. Instead of reducing arguments completely, they are only reduced until the expansion is possible. In order to do this the system needs to know which definition it is expanding to, therefore the definition identification in step 3 now occurs before this process.

The first step of the new process is to reduce any arguments which are causing a pattern matching ambiguity (section 3.1.1.1). Say that the application being reduced is `take (1+1) [square 1, square 2, square 3]` where `take` is defined as:

```

take :: Int -> [a] -> [a]
take _ [] = [] #Definition 1
take 0 (x:xs) = [] #Definition 2
take n (x:xs) = x : take (n-1) xs #Definition 3

```

The system has already determined which definition the function application will be expanded to, in this case definition 3. The process for determining which (if any) arguments need to be reduced is as follows:

1. The patterns belonging to the previous definitions are matched against the function arguments. In this case definitions 1 and 2 and the previous definitions.
2. If the arguments fail to match with all of the previous definition patterns then no ambiguity exists. Therefore the process can stop, and the program continue to dealing with any formal-actual mapping failures.
3. However if any of the patterns have the potential to match with the arguments then reduction is needed. In this case the pattern associated with definition 2 causes the ambiguity, as it has the potential to match the arguments.
4. The system needs to determine which of the arguments do in fact cause the ambiguity. An argument can only cause an ambiguity when its corresponding sub-pattern differs between definitions. In this case the second argument doesn't cause the ambiguity, as its corresponding sub-pattern is identical for both definition 2 and 3. Meanwhile the sub-patterns 0 and n differ, so the first argument (1+1) causes the ambiguity, and therefore needs to be reduced.
5. The first argument which needs to be reduced, is then reduced by a single step. The system then returns to step one to check if the ambiguity has been removed.

In the example all ambiguities are removed after one reduction step, namely 1+1 being reduced to 2. Now all ambiguities has been removed any formal-actual mapping failures must be dealt with (section 3.1.1.2).

1. Each sub-pattern and its respective argument are matched against one another. The sub-patterns come from the definition that is being expanded to, in this case definition 3. Here n is matched against 2, and x:xs is matched against [square 1, square 2, square 3].
2. If the match fails for any of the arguments, then that argument is reduced by a single step, and the system returns to step one.
3. Otherwise the arguments and sub-patterns all successfully match, so any formal-actual mapping failures have been dealt with. The formal-actual maps generated by this process are combined, and can then be used in the next stages of the reduction process.

Implementing this required a new formal-actual mapping system which can:

- Map arguments which are not fully reduced.
- Detect if an argument and pattern have the potential to match.

A new system which has these abilities is described in section 3.3.

This implementation of call-by-name does have a limitation. It does not ensure that arguments are reduced far enough such that any guards can be evaluated. This is further discussed in section 3.6.1.

3.1.3.2 Implementing sharing

Implementing sharing required modification of step 5. Now shared formal-actual pairs are introduced as let bindings instead of being substituted into the AST.

They are considered shared when:

- The actual is not fully reduced.
- The formal appears more than once in the function definition.

Functionality for checking if the actual is fully reduced was implemented in the original stupid computer.

Counting occurrences of a formal is done by traversing the AST of the function definition. However, care has to be taken regarding scope, specifically in the cases of list comprehensions, lambda abstractions and let statements. For example see the following function:

```
f x y = (let x = y * y in x + x) * x
```

Due to the let statement, `x` should only be considered as occurring once, as the occurrences of `x` within the body of the let statement are out of scope.

If the formal-actual pair is shared then a let statement defining that value is inserted around the function definition. As the variables defined by let statements are always indexed, for example `x` becomes `x_0`, the original variable in the definition must be substituted for the indexed version. This is to avoid the names of created variables clashing with one another, as could happen when reducing a recursive function. The implementation details for let statements are included in section 3.5.1. Formal-actual pairs which are not shared are substituted into the definition as normal.

A full description of how let statements are defined in the stupid computer can be found in section 3.5.1.

3.2 Improved user interface

In the original program the expression the user wanted to be evaluated had to be included in the input file, as in figure 3.1. As Haskell source files cannot contain bare expressions these input files were not valid Haskell, and so could not be loaded by GHCi. It also made it harder for users to experiment with multiple input expressions, as they would have to modify the input file for each.

```

1  map :: (a -> b) -> [a] -> [b]
2  map f (x:xs) = (f x) : (map f xs)
3  map _ [] = []
4
5  square :: Num a => a -> a
6  square x = x*x
7
8  --This is an example of using a higher order function (map)
9  map square [1..3]

```

Figure 3.1: Example of previous input file. The expression being evaluated here is `map square [1..3]`

Therefore the interface has been updated to be more similar to GHCi. The input file now contains a Haskell module, and the user provides expressions to evaluate via the interface. Once a reduction has finished the user can input a new expression to be reduced, or the user can input “:q” to exit the program. An example input file, and a screenshot of the system being used are in figure 3.2. Note that on line 3 the user has excluded the definition of `map` from the prelude. This is required to stop name clashes when redefining prelude functions, the reasons for this are explained further in section 3.5.5.

```

1  module Map where
2
3  import Prelude hiding (map)
4
5  map :: (a -> b) -> [a] -> [b]
6  map f (x:xs) = (f x) : (map f xs)
7  map _ [] = []
8
9  square :: Num a => a -> a
10 square x = x*x
11

```

(a) Example of new input file.

```

alexw@Alexander-Waseys-MacBook-Pro examples % stupid-computer map.hs
Environment = map.hs
map square [1..3]
  map square [1..3]
= (square 1) : (map square [2..3])
= (1 * 1) : (map square [2..3])
= 1 : (map square [2..3])
= 1 : ((square 2) : (map square [3]))
= 1 : ((2 * 2) : (map square [3]))
= 1 : (4 : (map square [3]))
= 1 : (4 : ((square 3) : (map square [])))
= 1 : (4 : ((3 * 3) : (map square [])))
= 1 : (4 : (9 : (map square [])))
= 1 : (4 : (9 : []))
= 1 : (4 : [9])
= 1 : ([4, 9])
= [1, 4, 9]

Environment = map.hs

```

(b) Evaluating `map square [1..3]` with the new interface.

Figure 3.2: Example of new input file, and its usage.

The system also provides error messages when the user gives an invalid expression, for example in figure 3.3 the user is trying to evaluate `foldr` with arguments in the incorrect order. As it was not possible to implement in-depth error messages the system directs the user to test their code with `GHCi`.

```
Environment = foldr.hs
foldr (+) [1,2,3,4] 0
Your code will not run, try checking it in GHCi!
Environment = foldr.hs
█
```

Figure 3.3: Error message seen on bad input

3.3 New formal-actual mapping system

While the original formal-actual mapping system was suitable for an initial implementation it had a number of limitations.

1. It required `GHC` interpretation upon each use, as use of the evaluator is quite expensive this slowed performance.
2. The system forced all arguments to be fully evaluated as they were mapped. For example mapping `(x:xs)` against `[square 1, square 2, square 3]` would ideally result in a mapping of `[("x" : square 1), ("xs" : [square 2, square 3])]`. However due to the call to `show` in the created function the arguments get fully evaluated, so the actual mapping created was `[("x" : 1), ("xs" : [4,9])]`. This specifically prevented the implementation of lazy evaluation, as it forced function arguments to be reduced before function applications.
3. It was not possible to map against infinite data structures. Mapping `x` against `[0..]` requires `show [0..]` to be evaluated, which takes an infinite amount of time and memory.
4. Attempting to match against functions was not possible, as the call to `show` required all arguments to be showable, which Haskell functions are not.

To address these issues a new system has been implemented which does not involve `GHC` interpretation. This system serves two purposes. Firstly the system can check to which extent a pattern and argument match. If they fully match it can then generate the formal-actual map for that pattern argument pair.

There are three possible outcomes when checking for a match:

- **Match** : The pattern and argument currently match against one another. For example `(x:xs)` and `[1,2,3]`.
- **Potential-Match** : The pattern and argument do not currently match, but have the potential to do so when the argument is further reduced. For example the pattern `(x:xs)` and the argument `map square [1,2,3]`.
- **No-Match** : The pattern and argument do not match, and cannot do so even if the argument is further reduced. For example `(x:xs)` and `[]`.

Both the checking and the map generation are achieved by recursively comparing the AST of the pattern against the AST of the argument. Below is a summary of how this works for each kind of pattern. Note that while literal, list, tuple and cons patterns are semantically special cases of constructor patterns, in the GHC AST they have different representations. Therefore they all need to be handled individually.

- **Pure formal patterns.** When the pattern is a pure formal, such as `x`, the pattern and argument always match. The mapping between the argument and the pattern can be simply added to the formal-actual map.
- **Wildcard patterns.** When a wildcard pattern is encountered (`_`), the pattern and argument again always match. However in this case there is nothing to add to the formal-actual map.
- **Literal patterns.** Literal values, such as `2` or `"Hello, World!"` can only match against fully reduced arguments. If the value of such a fully reduced argument matches that in the pattern then there is a match. In this case there is nothing to add to the formal-actual map.

If the argument is not fully reduced, then the system cannot yet know if it matches, therefore it is considered a potential match.

- **List patterns.** This is the case where the pattern is a list of patterns, such as `[1, b, c]`. These patterns can only match against arguments which have been reduced to list form, such as `[square 1, square 2, square 3]` or `[1,4,9]`. The number of elements in the pattern and argument lists must be the same, otherwise the match fails. If there is no discrepancy then the corresponding items in the pattern and argument are matched against one another. If any don't match, the entire list is considered to not match. Similarly if any only potentially match then the entire list is considered to be a potential match. But if all elements do match then the produced formal-actual maps are combined and returned.

In the other case where the argument is not in list form, but could be with further reduction, such as `map square [1,2,3]` then a potential-match is returned.

- **Tuple patterns.** Tuple patterns follow the same process as list patterns, however no length checks are required. This is because the length of tuples is fixed, any discrepancies will be caught by the validity checking stage (section 2.2.3).
- **Constructor patterns.** These are patterns such as `Nothing` or `Just a`. Again these can only match against arguments which have been reduced to a construc-

tor, such as `Just (1+1)`. First it must be checked that the constructor in the pattern and argument have the same name, otherwise no match is possible.

If the constructor has any arguments, then these are matched between the pattern and argument, if any fail to match then the entire constructor fails to match. If all these matches are successful then the produced formal-actual maps are combined and returned.

When the argument is not in constructor form then a potential match is returned.

- **As pattern.** The `as` pattern, such as in `x@(y:ys)` binds the argument to the identifier on the left hand side of the pattern, in this case `x`. It then matches the entire argument against the pattern on the right hand side, in this case `(y:ys)`. These two mappings are combined and returned. If the right hand matching fails the entire match fails.
- **Cons pattern.** The `cons` pattern `(x:xs)` is used to decompose lists. The item at the head of the list is matched against the left hand side of the `cons` (`x` in this case), while the tail is matched against the right hand side (`xs`). How the argument is decomposed differs slightly between arguments:
 - **Lists** A match is only possible if the list has at least one argument. The left hand side of the concatenation is matched against the head of the list, and the right hand side against the tail of the list.
 - **Strings** A match is only possible if the string is non-empty. The head of the string is converted to a character and matched against the left hand side, the rest of the string is matched against the right.
 - **Sequences** A sequence such as `[1..5]` is split into its head value (`1`) and a tail sequence (`[2..5]`). These are then matched against the left and right hand side patterns respectively.
 - **Concatenation function** Where the argument is an application of the concatenation function, such as `square 1 : map square [2,3]`, then the left hand side of the application is mapped against the left side of the pattern, and the right side of the application mapped to the right of the pattern.
 - **Other function applications** All other function applications, such as `map square [1,2,3]` cannot yet be decomposed, and therefore return a potential match.

The results of matching the left and right hand sides are combined, if either didn't match then the concatenation as a whole is considered to not match.

This system works effectively and resolves the issues which affected the older solution, allowing lazy evaluation to be implemented. However this system was far more complex to implement and test than the original system. This is because each individual case had to be accounted for, rather than relying on GHC. The relative complexity of the GHC AST compared to Haskell syntax, as mentioned in section 2.2.2, further complicated implementation.

3.4 Reparsing of GHC interpretation results

The GHC interpretation ability mentioned in section 2.2.5 was flawed in its original implementation. Because the type of the expression being evaluated is unknown at compile time, then the result from `hint` [The Hint Authors, 2007] is always a string. In the first version of the stupid computer this resulting string was placed within an `HsVar` node in the AST. This caused issues when results were data structures, as they cannot be properly dealt with by the rest of the stupid computer. This limited the ability of users to make use of functions from the prelude within their code. Say for example that `sum` has been user defined, but `map` has not, then when trying to reduce `sum (map (\x -> x*x) [1,2,3])` the reduction will fail.

```
sum (map (\x -> x*x) [1,2,3])
= sum [1,4,9]
```

The new pattern matching system expects lists to be held within `ExplicitList` nodes, not `HsVar` nodes. Therefore it cannot pattern match against the list contents and the reduction fails. Notably the old pattern matching system did not encounter this issue as it used GHC interpretation, however there were still problems in other contexts such as list comprehensions. To resolve this issue results from the interpreter are now reparsed by GHC, giving a properly formed AST which can then be substituted into the main expression tree. This has proved effective, allowing the previously failing trace to finish successfully:

```
sum (map (\x -> x * x) [1, 2, 3])
= sum ([1, 4, 9])
= 1 + sum [4, 9]
= 1 + 4 + sum [9]
= 1 + 4 + 9 + sum []
= 1 + 4 + 9 + 0
= 1 + 4 + 9
= 1 + 13
= 14
```

This gives users full flexibility to use prelude functions within their code.

3.5 Improved feature support

This section covers the Haskell features which now have proper support in the stupid computer.

3.5.1 Let statements

In section 3.1.2 let statements are inserted into to allow sharing. Let statements were not previously supported by the stupid computer, so support has been added in this iteration. It is important to note that Haskell let statements are equivalent to `letrecs` in other languages [Turner et al., 1995], which means the values defined by a let are visible when those values are being evaluated. For example when evaluating

`let x = 4 in let x = 2; y = 2 + x; in y` the value of the second `x` is visible when calculating `y`, giving a final result of 4.

The reduction semantics from the call-by-need lambda calculus have been adapted for a Haskell context [Maraist et al., 1998]. These are as follows:

3.5.1.1 Value

When a bound value is needed by the body, and the value is fully reduced, it can be substituted into the body of the `let`. For example:

```
let x = 4 in 2 + x
= let x = 4 in 2 + 4
```

Note that only one substitution happens upon each step, so in the case of `let x = 4 in x * x`:

```
let x = 4 in x * x
= let x = 4 in 4 * x
= let x = 4 in 4 * 4
```

3.5.1.2 Evaluate

When a value has not been fully reduced, but it is needed by the body then it will undergo a reduction step. For example in the expression `let x = sum [1,2,3] in x * x`:

```
let x = sum [1,2] in x * x
= let x = 1 + sum [2] in x * x
= let x = 1 + 2 + sum [] in x * x
= let x = 1 + 2 + 0 in x * x
= let x = 1 + 2 in x * x
= let x = 3 in x * x
...
```

This continues until the value is fully reduced, at which point it can be substituted into the `let` body with a value step.

3.5.1.3 Split

Where the bound value is needed, and is a concatenation, it can be split into two new variables. For example the expression `let x = map square [1,2,3] in sum x` is reduced as follows:

```
let x = map square [1,2,3] in sum x
= let x = (square 1) : map square [2,3] in sum x
= let x_0 = (square 1) in
  let x_1 = map square [2,3] in let x = (x_0:x_1) in sum x
...
```

This rule is not included in the call-by-need calculus, but has been added to allow the evaluation of lists to be shared, even if they are infinite.

3.5.1.4 Garbage Collection

When the variable being bound no longer appears in the body of the let expression it can be removed. For example:

```
let x = 4 in 2 + x
= let x = 4 in 2 + 4
= 2 + 4
= 6
```

Garbage collection does not change the semantics of reduction, the intent is instead to keep expressions as simple as possible for the user.

3.5.1.5 A value being ‘needed’

A value within a let expression is considered ‘needed’ when the body of the let expression cannot be further reduced without it. For example in the following reduction x only becomes needed once nothing else in the expression can be reduced.

```
let x = 4 in x + (2 * 3)
= let x = 4 in x + 6
= let x = 4 in 4 + 6
= 4 + 6
= 10
```

This property delays the reduction of bound let values until the last possible moment, this is especially important in circumstances where the bound value may never be used. This can be illustrated with the example of `let x = sum [1..] in if (2 < 3) then 0 else x`, if the bound value is evaluated when not needed then the resulting reduction is:

```
let x = sum [1..] in if (2 < 3) then 0 else x
= let x = 1 + 2 + sum [3..] in if (2 < 3) then 0 else x
= let x = 1 + 2 + 3 + sum [4..] in if (2 < 3) then 0 else x
= let x = 1 + 2 + 3 + 4 + sum [5..] in if (2 < 3) then 0 else x
...
```

This will never halt, however with the proper semantics.

```
let x = sum [1..] in if (2 < 3) then 0 else x
= let x = sum [1..] in if (True) then 0 else x
= let x = sum [1..] in 0
= 0
```

The unnecessary calculation is avoided, and the reduction can complete.

3.5.1.6 Let expression creation

When a let expression is created the values being bound by the let are given index numbers. For example say that map is defined as follows:

```
map f (x:xs) = let y = f x in y : (map f xs)
map f [] = []
```

Then the expansion of `map square [1,2,3]` will result in:

```
map square [1,2,3]
= let y_0 = square 1 in y_0 : (map square [2,3])
= let y_0 = square 1 in
  y_0 : (let y_1 = square 2 in y_1 : (map square [3]))
...
```

Here `y` is replaced with `y_0` when the first let is created, and in the next step `y_1` replaces `y`. This is primarily done to avoid name clashes when reducing recursively defined functions, which is especially important as Haskell lets are letrecs [Turner et al., 1995].

3.5.2 Lambda abstractions

Proper support for lambda abstractions, sometimes known as anonymous functions, have been added to the stupid computer. Reducing the application of such an abstraction is as follows:

```
(\x -> x * x) 3
= 3 * 3
= 9
```

Full support for pattern matching is also included:

```
(\ (x:_) -> x) [1,2,3]
= 1
```

Much like named functions lambda abstractions use lazy evaluation, however in the case of a pattern matching failure arguments are still evaluated as far as needed:

```
(\ (x:)) -> x (map square [1,2,3])
= (\ (x:_) -> x) (square 1) : (map square [2,3])
= square 1
= 1 * 1
= 1
```

This support for laziness also includes support for the sharing of values, much like named functions.

```
(\x -> x * x) (1+2)
= let x_0 = (1+2) in x_0 * x_0
= let x_0 = 3 in x_0 * x_0
= let x_0 = 3 in 3 * x_0
```

```

= let x_0 = 3 in 3 * 3
= 3 * 3
= 9

```

Adding this functionality was made simpler by utilising functionality implemented earlier in the project. The process to reduce a lambda abstraction is as follows:

1. The system first determines if the argument needs any further reduction. It does this by trying to match the argument against the pattern on the left hand side of the abstraction. This is done with the formal-actual mapping system described in section 3.3. If they do not match then the argument is reduced by a single step, and the system tries the matching again. If they do match then the reduction of the abstraction can continue, making use of the formal-actual map produced.
2. The formals in the abstraction body are substituted for the actuals in the formal-actual map. This is done with the process described in section 3.1.3.2. This is to ensure that the appropriate let statements are inserted, such that lazy evaluation is utilised.
3. The body of the abstraction can then be substituted into the expression.

3.5.3 Case statement support

Support has been added for case statements. As an example if `sum` is defined as:

```

sum xs = case xs of
  (y:ys) -> y + sum ys
  [] -> 0

```

Then the full reduction of `sum (map square [1,2])` is:

```

sum (map square [1, 2])
= case (map square [1, 2]) of
  (y : ys) -> y + sum ys
  [] -> 0
= case ((square 1) : (map square [2])) of
  (y : ys) -> y + sum ys
  [] -> 0
= (square 1) + sum (map square [2])
= (1 * 1) + sum (map square [2])
= 1 + sum (map square [2])
= 1 + case (map square [2]) of
  (y : ys) -> y + sum ys
  [] -> 0
= 1 + case ((square 2) : (map square [])) of
  (y : ys) -> y + sum ys
  [] -> 0
= 1 + (square 2) + sum (map square [])
= 1 + (2 * 2) + sum (map square [])
= 1 + 4 + sum (map square [])

```

```

= 1 + 4 + case (map square []) of
      (y : ys) -> y + sum ys
      [] -> 0
= 1 + 4 + case [] of
      (y : ys) -> y + sum ys
      [] -> 0
= 1 + 4 + 0
= 1 + 4
= 5

```

Much like function applications the argument to the case statement is reduced until there is no ambiguity or pattern matching failure. Case statements also make use of lazy evaluation.

```

case [1*1, 2, 3] of
  (y : ys ) -> y + y
  [] -> 0
= let y_0 = 1*1 in y_0 + y_0
= let y_0 = 1 in y_0 + y_0
= let y_0 = 1 in 1 + y_0
= let y_0 = 1 in 1 + 1
= 1 + 1
= 2

```

Fortunately it was relatively simple to implement support for case statements. This is because I was able to leverage the functionality I implemented for function application reductions. This implementation is as follows:

1. Firstly the definition to be reduced to must be determined. This is implemented using the pattern matching system described in section 3.3. The program looks at each possible definition in turn:
 - (a) If the argument and pattern do match, then this definition is selected. The formal-actual mapping produced is retained.
 - (b) Otherwise if they only have the potential to match then there is an ambiguity, therefore the definition to reduce to cannot yet be determined.
 - (c) If the argument and pattern do not match, or have the potential to match, then move onto the next pattern and repeat from step one. However if no more patterns remain then the correct definition cannot be determined.
2. If the definition to expand to cannot yet be determined, then there is an ambiguity or pattern matching failure. Therefore the argument is reduced by a step, after which this process begins again from step one. Otherwise if the definition has been determined then the process can continue to step three.
3. The formals in the definition are substituted for the actuals in the formal-actual map (generated in step one). This is done with the new process described in

section 3.1.3.2. This means that let expressions are inserted where needed, such that lazy evaluation is used in this context.

3.5.4 Custom infix operator support

Support has been added for user defined infix functions, for example below is a possible definition for the * function.

```
(*) :: Int -> Int -> Int
m * 0 = 0
m * n = (m * (n-1)) + m
```

The trace for 5 * 3 is therefore:

```
5 * 3
= (5 * (3 - 1)) + 5
= (5 * 2) + 5
= ((5 * (2 - 1)) + 5) + 5
= ((5 * 1) + 5) + 5
= (((5 * (1 - 1)) + 5) + 5) + 5
= (((5 * 0) + 5) + 5) + 5
= ((0 + 5) + 5) + 5
= (5 + 5) + 5
= 10 + 5
= 15
```

This has been implemented by converting infix function definitions to prefix definitions at the initial parsing stage. When infix function applications are reduced they are first converted to prefix applications, reduced, and then converted back to infix form. This allowed the leveraging of existing functionality for reducing prefix functions, rather than developing a separate implementation.

3.5.5 Support for user defined data types

Support has been added for user defined data types, these are treated semantically just like inbuilt data types. As an example given the following definition of a new type, and a function on it:

```
data Sometimes a = Contains a | Empty

getValue (Contains i) = i
getValue Empty = -1
```

Then the following reduction on this type (making use of pattern matching) work as expected.

```
getValue Empty
= -1

getValue (Contains 3)
```


= 3

Supporting user defined data types was not possible in the initial version of the stupid computer, due to GHC interpretation. In some situations, such as when checking guard conditions, Hint needs access to the user's definitions. In the initial implementation (section 2.2.5) of GHC interpretation let expressions were used to provide function definitions to the interpreter. However it is not possible to place type definitions in let expressions. This prevented GHC interpretation of any function which made use of a user defined type.

I have fixed this issue for this version of the tool. Instead of providing the definitions using a let expression, now the interpreter loads the users input file directly as a module. The main challenge faced when implementing this was preventing clashes between a users definitions and the prelude. For example if the users module contains a definition of `sum`, then the interpreter does not know if uses of `sum` in the code refer to the prelude definition or the users. Fortunately Hint allows functions in loaded modules to be excluded from the prelude. This has allowed me to resolve the issue by providing Hint with a list of functions which have been defined by the user. Hint then ignores the versions of these functions in the prelude.

However this solution has two drawbacks for users.

- Users must format their input files as modules. However it should be relatively simple for users to deal with this restriction.
- Users need to also exclude the prelude functions they redefine in their own source files. For example if the user has redefined `sum` they must include the following line in their source file:

```
import Prelude hiding (sum)
```

This is required because when Hint first loads the users module, it implicitly loads Prelude without any exclusions. Therefore users must include this line to allow the interpreter to load their source file. However as users already need to include this line to compile files which redefine prelude functions it is not a large issue.

3.6 Current system limitations

This section describes the current limitations to the system, and discusses some possible solutions.

3.6.1 Failure to evaluate guard values

Currently the system does not take into account how far arguments need to be reduced for guards to be evaluated. Taking the following function definition as an example:

```
gt x (y:ys) | y > x = y : (gt x ys)
            | otherwise = gt x ys
```

```
gt _ [] = []
```

If `gt 1 [1+1, 1+2, 1+3]` is being reduced then the correct reduction would be:

```
gt 2 [1+1, 1+2]
= gt 2 [2, 1+2]
= gt 1 [1+2]
= gt 1 [3]
= 3 : gt 1 []
= 3 : []
= [3]
```

The additions in the list are reduced before the function, as their values are needed to determine the outcome of the guards. However my implementation does not account for this, therefore it gives the following trace:

```
gt 2 [1+1, 1+2]
= gt 2 [1+2]
= 1+2 : gt 1 []
= 3 : gt 1 []
= 3 : []
= [3]
```

Here the calculation of `1+1` is not shown to the user. This stems from the decision to not show the calculation of guards in section 3.1.3.1. To deal with this issue the system the stupid computer itself would need to evaluate guard conditions, instead of leaving it to GHC. This could be done by evaluating each guard condition in turn, until one evaluates as being true. This would force the reduction of the arguments up to the point where the guard can be evaluated. Unfortunately due to time constraints I have not been able to implement this solution.

3.6.2 Lack of support for where statements

Support for where statements has not been included in this version of the stupid computer. Adding support is complicated as Haskell does not support their use within expressions, only within function definitions. This raises a couple of problems:

- A large aim of the project is to ensure that each output expression can be evaluated by the user with GHCi. However if the system were to include where statements in expressions this would no longer be possible. For example if the `map` function was defined using where :

```
map _ [] = []
map f (x:xs) = x' : (map f xs)
  where
    x' = f x
```

Then the trace would be:

```
map square [1,2,3]
```

```
= x' : (map square [2,3]) where x' = square 1
```

Attempting to evaluate the second expression with GHCi will result in a compiler error.

- Secondly the GHC AST does not actually allow for `where` expressions to be placed within expressions. Therefore to represent such expressions I would have to modify GHC to allow them to be constructed. While this would be possible, it would make it more difficult to keep the stupid computer up to date with newer compiler versions.

One possible solution to this issue would be to replace `where` statements with `let` expressions when expanding functions which use `where`. In the previous `map` example a trace would be:

```
map square [1,2,3]
= let x' = square 1 in x' : (map square [2,3])
= let x' = 1*1 in x' : (map square [2,3])
...
```

This solution would allow the user to use `where` statements, while keeping each expression valid Haskell. However due to the complexity of the GHC AST this solution would be fairly time consuming to implement. It is also possible I could encounter further issues which may make this solution unworkable. As it is relatively easy for students to convert their `where` statements to `let` statements I have decided to not add support.

3.6.3 Incorrect operator precedence

Currently the system does not properly respect operator precedence. As an example see the following trace:

```
2 + 3 * 4
= 5 * 4
= 20
```

In this example `+` has been given precedence over `*`. Of course `*` has precedence over `+`, so therefore the correct trace should be:

```
2 + 3 * 4
= 2 + 12
= 14
```

This is caused because GHC parses all infix operators as being left-associative. This means `2 + 3 * 4` is parsed as `(2 + 3) * 4` rather than `2 + (3 * 4)` as would be correct. In GHC these precedence issues are resolved by the renamer [Marlow et al., 2004], however the main purpose of the renamer is fully resolving identifiers, not dealing with operator precedences. So for example `sum` becomes `Data.Foldable.sum` and `+` becomes `GHC.Num.+`. Based on my initial investigations applying the precedence re-

association does not seem to be possible separately from the rest of the renaming stage. I considered two approaches to resolve this issue.

- Applying the entire renaming step to the AST and then undoing the identifier resolution. This would avoid having to implement and test an implementation of the re-association functionality which already exists in GHC. However this turned out to be infeasible, as the renamer makes large changes to the structure of the AST. For example in the output of the parser each function is represented by a `ValD` node, these are within a `HsModule` node. In comparison in the output of the renamer these are within a `HsGroup` node. This would require modifying large sections of my code which deal with this part of the AST. It is also possible that other parts of the AST structure are modified, but further investigation is needed to determine this. This makes it hard to predict how much work would be required to modify my code to make use of the renamer.
- Therefore the better option is to implement my own version of this functionality, possibly based on the approach GHC takes. While the functionality itself would require implementation and testing, not having to modify the rest of the code base makes it the better option. I estimated that this would take around a 10 hours of work to implement and test, however it is likely I would encounter further issues that would make this an underestimate. Therefore since it is relatively easy to work around this issue by asking the user to add parenthesis I decided not to tackle the issue.

Chapter 4

Evaluation

4.1 Testing

The stupid computer has been tested with a set of code samples, which were also used to test the original implementation. These samples were taken from the slides and quizzes of the course Informatics-1A [Wadler et al., 2022], to ensure they are representative of code students are likely to write. It would have been impractical to use all code samples from the course, therefore the set of cases used focus on samples presented in the context of a stupid computer trace. Efforts have been made to avoid repeating similar samples. Code samples have also been modified to avoid `where` statements as they are not explicitly supported. However I have included one test with a `where` statement, to test if the program manages to reduce such statements directly to their results, as seen in section 2.1.3.5. Alongside these additional tests have been written by myself to try and attain better coverage. Overall there were 46 code samples, of which 31 were taken from Informatics-1A. When previously tested 11 samples failed to run, and five ran but had incorrect results. This new version has seen a large improvement, with one test which failed to run, and one ran but with incorrect results. The failure was due to a lack of support for `where` statements, as discussed in section 3.6.2. The incorrect result was caused by the incorrect operator precedences, as seen in section 3.6.3.

The tests, alongside their stupid computer trace, can be found in `tests/` in the project directory. The traces from the first year version of the stupid computer are also included for comparison.

4.2 User Feedback

To help inform development feedback has been sought from potential users. This was done via a survey, sent to students who had recently completed the Inf1A course. It was split into two parts, one asking their views on the concept, and the other asking them to install and use the tool. The second part was optional. This reduced the time requirement to complete the survey, thus allowing for a larger number of re-

sponses. This study was certified according to the Informatics Research Ethics Process RT 2019/53817. The participant information sheet is included in appendix A.3.

4.2.1 Part one

Users were shown a summary of the stupid computer, as can be seen in appendix A.1, then asked the following questions.

1. How often do you think you would use this tool? [Daily, Weekly, Never, Other]
2. Do you have any further thoughts on the tool?

I received 37 responses to question one, and the results are summarised in figure 4.1. There were five response of daily, 23 responses of weekly and 4 responses of never. Of the five other responses 4 gave more specific times they would use it, such as when debugging, and one user responded they didn't use Haskell. This means that of the 37 respondents around 85% could foresee themselves using the program at least semi-regularly. This shows a strong enthusiasm for the stupid computer concept among the target user base.

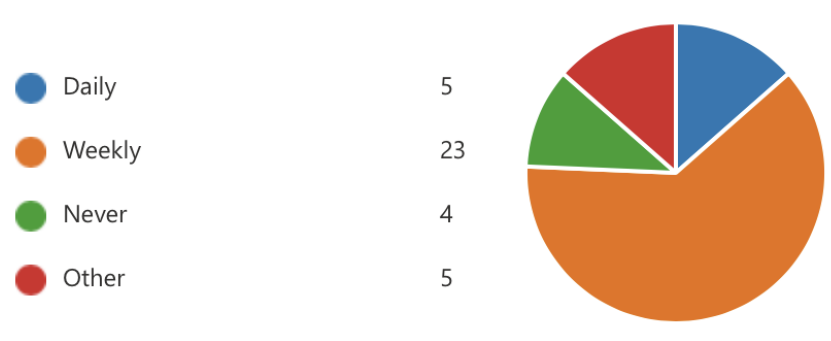


Figure 4.1: Survey question one results.

Question two allowed open responses, of which I received 21. The majority of the comments were positive in nature, with most saying that they thought the program would be useful for novice programmers. There were also many comments stating that the stupid computer needs to be easy to install and use, along with suggestions that it could be a web application.

4.2.2 Part two

The second part of the survey involved asking users to download and run the stupid computer on their own machines. They were provided installation instructions and asked the following questions. See appendix A.2 for the full instructions.

- Were you able to install the stupid computer using these instructions?
- What, if any, difficulties did you face?
- Were you able to use the stupid computer?

- What, if any, difficulties did you face?
- Do you think the tool is useful?
- Do you have any comments about using the tool?

Eight people continued to this section of the survey, and unfortunately only 50% of them were able to install the stupid computer using the instructions provided. Issues with the process included the instructions being hard to follow, users not having stack installed, and a lack of dynamic compilation support on windows. This last issue has been resolved by changing the project from dynamically to statically compiled binaries. During early development of the stupid computer there was an issue with the hint [The Hint Authors, 2007] library which prevented static builds, but this appears to be resolved. The first two issues can be resolved by making prebuilt executables available for each of the major operating systems, which students can then download and run.

Of the four people able to complete the install process three were able to use the stupid computer. In the case of the user unable to do so it appeared to be due to a zsh configuration issue, this should also be resolved by directly distributing the executables. All users who were able to run the tool thought that it was useful, and there were no new issues raised in response to the final question.

Overall I can conclude that while students consider the tool to be useful, its ease of use will be necessary to its success.

Chapter 5

Conclusions

The main goal of this half of the project, implementing lazy evaluation, has been achieved. Both aspects of laziness, call-by-name and sharing, are demonstrated to users. Call-by-name evaluation properly delays evaluation of values until they are needed. Meanwhile let statements are inserted where needed, to demonstrate the sharing of values. However a limitation of this implementation is that the evaluation of values needed by guards is not shown to the user. This issue, and a possible solution to it, is discussed in section 3.6.1. Implementing lazy evaluation required adding support for let expressions, a new formal-actual mapping system, and improvements to GHC interpretation.

Proper support has been added for case statements and lambda abstractions. Users can also define their own data types, and their own infix operators. However there is still a lack of support for where statements, and issues with operator precedence. Fixing these issues, as discussed in sections 3.6.2 and 3.6.3, would be my priority for future work.

Testing of the final program has shown a reduction in the number of tests failed, from 16 with the original implementation, to two with this new implementation. Feedback was also sought from students, with 37 responses, with a generally positive response. However students raised concerns about how easily the program can be installed and used. This showed that students were likely to use the stupid computer if it is made available to them. Therefore making access to the program as easy as possible would be another valuable future development.

Bibliography

- Friedman, D. P., Wise, D. S., and Wand, M. (1976). Recursive programming through table look-up. In *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, pages 85–89.
- Henderson, P. and Morris Jr, J. H. (1976). A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*, pages 95–103.
- Jones, S. P. (2003). *Haskell 98 language and libraries: the revised report*. Cambridge University Press.
- Lipovaca, M. (2011). *Learn you a haskell for great good!: a beginner's guide*. no starch press.
- Maraist, J., Osersky, M., and Wadler, P. (1998). The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317.
- Marlow, S., Jones, S. P., et al. (2004). The glasgow haskell compiler.
- Sannella, D., Fourman, M., Peng, H., and Wadler, P. (2022). *Introduction to Computation: Haskell, Logic and Automata*. Undergraduate Topics in Computer Science. Springer International Publishing, 1 edition.
- The GHC Team (2001). Prelude. <https://hackage.haskell.org/package/base-4.14.1.0/docs/Prelude.html>. Accessed on 09.03.2021.
- The GHC Team (2002). The ghc api. <https://hackage.haskell.org/package/ghc>. Accessed on 05.10.2020.
- The Hint Authors (2007). hint. <https://hackage.haskell.org/package/hint>. Accessed on 05.10.2020.
- Turner, D. A. (2016). Recursion equations as a programming language. In Lindley, S., McBride, C., Trinder, P., and Sannella, D., editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 459–478. Springer International Publishing, Cham.
- Turner, D. N., Wadler, P., and Mossin, C. (1995). Once upon a type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 1–11.

Wadler, P., Bradfield, J., and Fourman, M. (2022). Informatics 1a - university of edinburgh. <http://www.drps.ed.ac.uk/21-22/dpt/cxinfr08025.htm>. Accessed on 21.09.2020.

Wasey, A. (2021). Bringing the stupid computer to life.

Appendix A

Additional survey documents

A.1 Stupid Computer Overview

The stupid computer is a tool emulating the Haskell traces you saw in lectures. In operation a trace of the sum function looks like this:

```
alexw@Alexanders-MacBook-Pro-3 stupid_computer % stupid-computer examples/sum.hs
Environment = examples/sum.hs
sum [1,2,3,4]
  sum [1, 2, 3, 4]
  = 1 + sum [2, 3, 4]
  = 1 + 2 + sum [3, 4]
  = 1 + 2 + 3 + sum [4]
  = 1 + 2 + 3 + 4 + sum []
  = 1 + 2 + 3 + 4 + 0
  = 1 + 2 + 3 + 4
  = 1 + 2 + 7
  = 1 + 9
  = 10
Environment = examples/sum.hs
```

Where sum.hs looks like this

```
sum :: Num a => [a] -> a
sum (x:xs) = x + sum xs
sum [] = 0
```

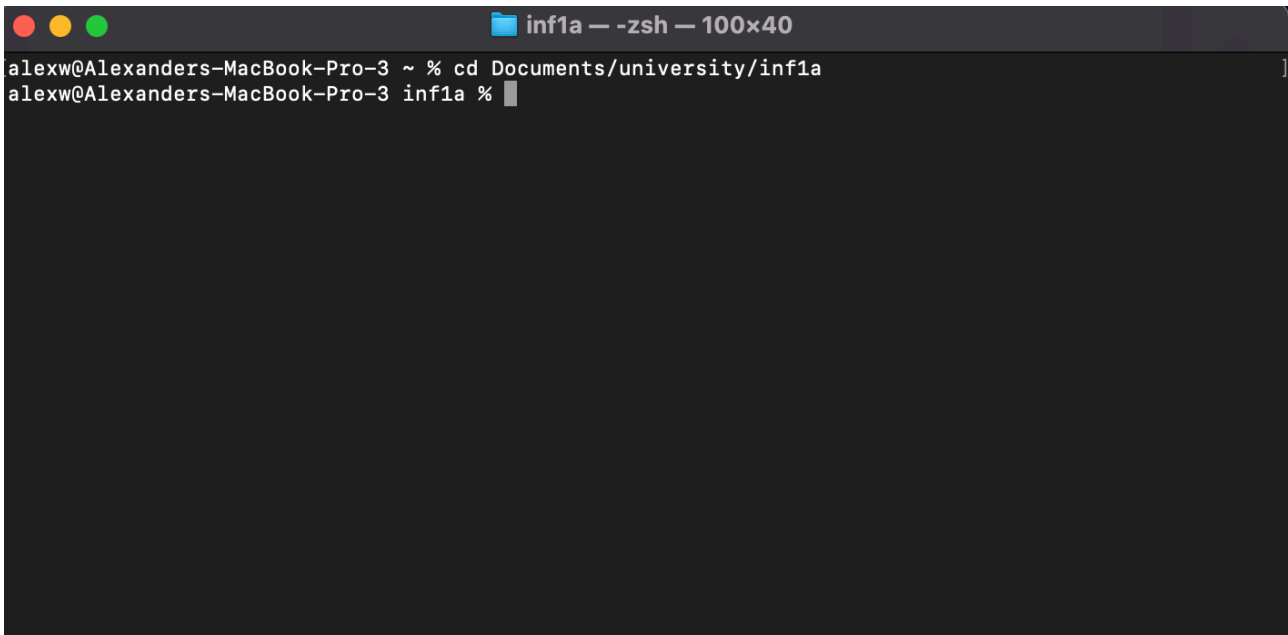
As another example the evaluation of a list comprehension works like this:

```
[x*x | x <- [1,2,3,4], x > 2]
= [x * x | x <- [1, 2, 3, 4], x > 2]
= [1 * 1 | 1 > 2] ++ [x * x | x <- [2, 3, 4], x > 2]
= [1 * 1 | False] ++ [x * x | x <- [2, 3, 4], x > 2]
= [] ++ [x * x | x <- [2, 3, 4], x > 2]
= [] ++ [2 * 2 | 2 > 2] ++ [x * x | x <- [3, 4], x > 2]
= [] ++ [2 * 2 | False] ++ [x * x | x <- [3, 4], x > 2]
= [] ++ [] ++ [x * x | x <- [3, 4], x > 2]
= [] ++ [] ++ [3 * 3 | 3 > 2] ++ [x * x | x <- [4], x > 2]
= [] ++ [] ++ [3 * 3 | True] ++ [x * x | x <- [4], x > 2]
= [] ++ [] ++ [3 * 3] ++ [x * x | x <- [4], x > 2]
= [] ++ [] ++ [9] ++ [x * x | x <- [4], x > 2]
= [] ++ [] ++ [9] ++ [4 * 4 | 4 > 2]
= [] ++ [] ++ [9] ++ [4 * 4 | True]
= [] ++ [] ++ [9] ++ [4 * 4]
= [] ++ [] ++ [9] ++ [16]
= [] ++ [] ++ [9, 16]
= [] ++ [9, 16]
= [9, 16]
```

A.2 Installation guide

Installing the stupid computer

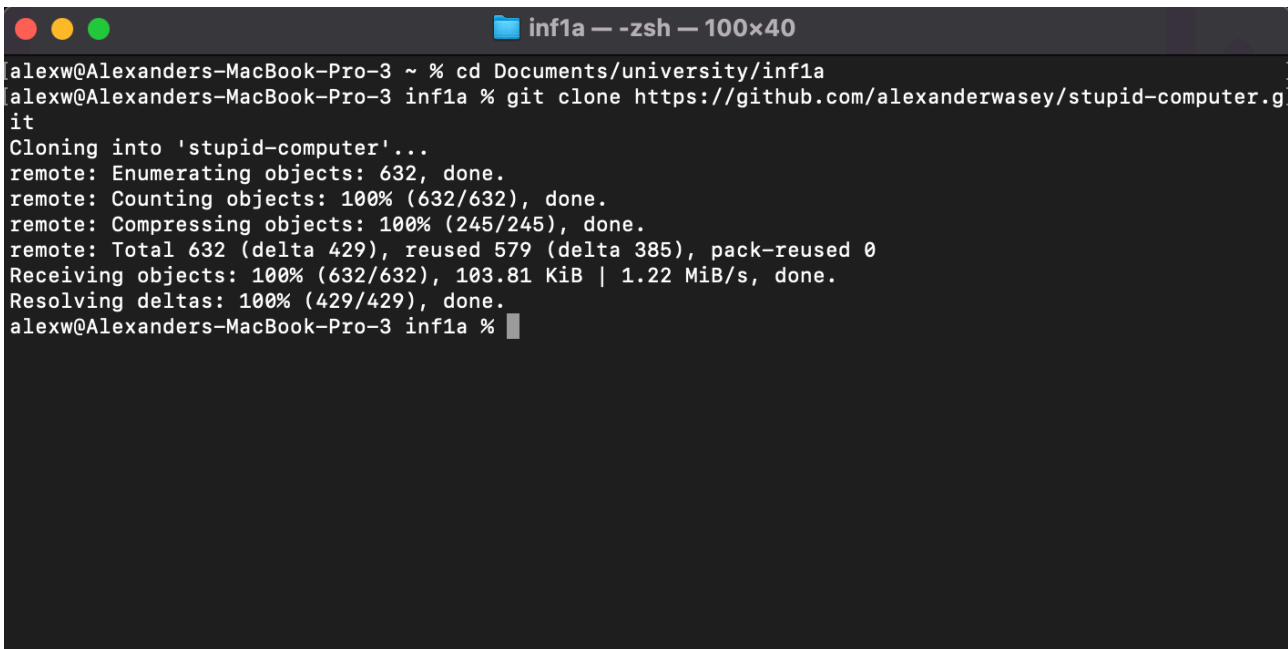
First open the terminal, and navigate to a suitable directory (for example your inf1a directory)

A terminal window titled 'inf1a — -zsh — 100x40' on a dark background. The prompt is 'alexw@Alexanders-MacBook-Pro-3 ~ %'. The user enters 'cd Documents/university/inf1a' and the prompt changes to 'alexw@Alexanders-MacBook-Pro-3 inf1a %'.

```
alexw@Alexanders-MacBook-Pro-3 ~ % cd Documents/university/inf1a
alexw@Alexanders-MacBook-Pro-3 inf1a %
```

Then run the following command:

```
git clone https://github.com/alexanderwasey/stupid-computer.git
```

A terminal window titled 'inf1a — -zsh — 100x40' on a dark background. The prompt is 'alexw@Alexanders-MacBook-Pro-3 ~ %'. The user enters 'cd Documents/university/inf1a' and the prompt changes to 'alexw@Alexanders-MacBook-Pro-3 inf1a %'. The user then enters 'git clone https://github.com/alexanderwasey/stupid-computer.git'. The terminal shows the output of the command, including progress bars for enumerating, counting, and compressing objects, and receiving deltas.

```
alexw@Alexanders-MacBook-Pro-3 ~ % cd Documents/university/inf1a
alexw@Alexanders-MacBook-Pro-3 inf1a % git clone https://github.com/alexanderwasey/stupid-computer.git
Cloning into 'stupid-computer'...
remote: Enumerating objects: 632, done.
remote: Counting objects: 100% (632/632), done.
remote: Compressing objects: 100% (245/245), done.
remote: Total 632 (delta 429), reused 579 (delta 385), pack-reused 0
Receiving objects: 100% (632/632), 103.81 KiB | 1.22 MiB/s, done.
Resolving deltas: 100% (429/429), done.
alexw@Alexanders-MacBook-Pro-3 inf1a %
```

This will download the source code for the stupid computer to a new directory, sensibly called 'stupid-computer'

Navigate to this directory with the command

```
cd stupid-computer
```

```
stupid-computer -- -zsh -- 100x40
alexw@Alexanders-MacBook-Pro-3 ~ % cd Documents/university/inf1a
alexw@Alexanders-MacBook-Pro-3 inf1a % git clone https://github.com/alexanderwasey/stupid-computer.g
it
Cloning into 'stupid-computer'...
remote: Enumerating objects: 632, done.
remote: Counting objects: 100% (632/632), done.
remote: Compressing objects: 100% (245/245), done.
remote: Total 632 (delta 429), reused 579 (delta 385), pack-reused 0
Receiving objects: 100% (632/632), 103.81 KiB | 1.22 MiB/s, done.
Resolving deltas: 100% (429/429), done.
alexw@Alexanders-MacBook-Pro-3 inf1a % cd stupid-computer
alexw@Alexanders-MacBook-Pro-3 stupid-computer %
```

You can now install the tool with the following command. This may take a while!

stack install

```
alexw@Alexanders-MacBook-Pro-3 stupid-computer % stack install
Building all executables for `stupid-computer' once. After a successful build of all of them, only s
pecified executables will be rebuilt.
stupid-computer> configure (exe)
Configuring stupid-computer-0.1.0.0...
stupid-computer> build (exe)
Preprocessing executable 'stupid-computer' for stupid-computer-0.1.0.0..
Building executable 'stupid-computer' for stupid-computer-0.1.0.0..
[1 of 9] Compiling ScTypes
[2 of 9] Compiling Tools
[3 of 9] Compiling PrepStage
[4 of 9] Compiling NormalFormReducer
[5 of 9] Compiling FormalActualMap
[6 of 9] Compiling DefinitionGetter
[7 of 9] Compiling EvalStage
[8 of 9] Compiling TypeCheck
[9 of 9] Compiling Main
Linking .stack-work/dist/x86_64-osx/Cabal-3.2.1.0/build/stupid-computer/stupid-computer ...
stupid-computer> copy/register
Installing executable stupid-computer in /Users/alexw/Documents/university/inf1a/stupid-computer/.st
ack-work/install/x86_64-osx/2f5dac9eef7782288931f88339055ffffb6c124dc8a97f7b3cb5223d2984bb782/8.10.4/
bin
Copying from /Users/alexw/Documents/university/inf1a/stupid-computer/.stack-work/install/x86_64-osx/
2f5dac9eef7782288931f88339055ffffb6c124dc8a97f7b3cb5223d2984bb782/8.10.4/bin/stupid-computer to /User
s/alexw/.local/bin/stupid-computer

Copied executables to /Users/alexw/.local/bin:
- stupid-computer
alexw@Alexanders-MacBook-Pro-3 stupid-computer %
```

This output may look quite different on your machine, don't panic!

You should now be able to run the stupid computer, some sample input files can be seen in the examples/ directory. We will run the sum example in this case. First launch the tool with the right file with the following command.

stupid-computer examples/sumpattern.hs

The tool should launch and look like this.

```
alexw@Alexanders-MacBook-Pro-3 stupid_computer % stupid-computer examples/sum.hs
Environment = examples/sum.hs
```

From here we can see the evaluation of the sum function, simply by giving an input expression. In this case we will input

```
sum [1,2,3,4]
```

And we can see the evaluation

```
alexw@Alexanders-MacBook-Pro-3 stupid_computer % stupid-computer examples/sum.hs
Environment = examples/sum.hs
sum [1,2,3,4]
  sum [1, 2, 3, 4]
= 1 + sum [2, 3, 4]
= 1 + 2 + sum [3, 4]
= 1 + 2 + 3 + sum [4]
= 1 + 2 + 3 + 4 + sum []
= 1 + 2 + 3 + 4 + 0
= 1 + 2 + 3 + 4
= 1 + 2 + 7
= 1 + 9
= 10

Environment = examples/sum.hs
```

Remember you must define in the input file the functions you want to see the evaluation for. Modifying the files in examples/ is a good place to start!

Uninstalling the Stupid Computer

To uninstall the tool you must first run

```
stack path --local-bin
```

This will tell you where cabal installed the tool, in my case

```
/Users/alexw/.local/bin
```

```
alexw@Alexanders-MacBook-Pro-4 stupid_computer % stack path --local-bin
/Users/alexw/.local/bin
alexw@Alexanders-MacBook-Pro-4 stupid_computer %
```

Calling this result **X** the stupid computer can be removed by running

```
rm X/stupid-computer
```

```
alexw@Alexanders-MacBook-Pro-4 stupid_computer % rm /Users/alexw/.local/bin/stupid-computer
alexw@Alexanders-MacBook-Pro-4 stupid_computer %
```


A.3 Participant Information Sheet

Participant Information Sheet

Project title:	Bringing the Stupid Computer to life
Principal investigator:	Phillip Wadler
Researcher collecting data:	Alexander Wasey

This study was certified according to the Informatics Research Ethics Process, RT number 2021/53817. Please take time to read the following information carefully. You should keep this page for your records.

Who are the researchers?

Alexander Wasey and Philip Wadler

What is the purpose of the study?

To evaluate the utility of the Stupid Computer tool to undergraduate students, to inform its future development.

Why have I been asked to take part?

You have taken part in Inf1A - Functional Programming

Do I have to take part?

No – participation in this study is entirely up to you. You can withdraw from the study at any time, up until February 2022 without giving a reason. After this point, personal data will be deleted and anonymised data will be combined such that it is impossible to remove individual information from the analysis. Your rights will not be affected. If you wish to withdraw, contact the PI. We will keep copies of your original consent, and of your withdrawal request.

What will happen if I decide to take part?

In the first part of the survey you will see be shown images of a tool designed to help students with understanding of functional programming concepts. You will then be asked to give your thoughts about the concept.

There is an optional second part of the survey, where you will be asked to download and run this tool, and then asked about your experiences.

You may be contacted for a follow up interview if you consent.

Are there any risks associated with taking part?



There are no significant risks associated with participation.

Are there any benefits associated with taking part?

You will be entered in draw to receive a £15 gift voucher for the Lighthouse Bookshop. (Redeemable in store or online)

What will happen to the results of this study?

The results of this study may be summarised in published articles, reports and presentations. Quotes or key findings will be anonymized: We will remove any information that could, in our assessment, allow anyone to identify you. With your consent, information can also be used for future research. Your data may be archived for a maximum of 4 years. All potentially identifiable data will be deleted within this timeframe if it has not already been deleted as part of anonymization.

Data protection and confidentiality.

Your data will be processed in accordance with Data Protection Law. All information collected about you will be kept strictly confidential. Your data will be referred to by a unique participant number rather than by name. Your data will only be viewed by the research team.

All electronic data will be stored on a password-protected encrypted computer, on the School of Informatics' secure file servers, or on the University's secure encrypted cloud storage services (DataShare, ownCloud, or Sharepoint) and all paper records will be stored in a locked filing cabinet in the PI's office. Your consent information will be kept separately from your responses in order to minimise risk.

What are my data protection rights?

Alexander Wasey is a Data Controller for the information you provide. You have the right to access information held about you. Your right of access can be exercised in accordance Data Protection Law. You also have other rights including rights of correction, erasure and objection. For more details, including the right to lodge a complaint with the Information Commissioner's Office, please visit www.ico.org.uk. Questions, comments and requests about your personal data can also be sent to the University Data Protection Officer at dpo@ed.ac.uk.

Who can I contact?



If you have any further questions about the study, please contact the lead researcher, Philip Wadler : philip.Wadler@ed.ac.uk

If you wish to make a complaint about the study, please contact inf-ethics@inf.ed.ac.uk. When you contact us, please provide the study title and detail the nature of your complaint.

Updated information.

If the research project changes in any way, an updated Participant Information Sheet will be made available on <http://web.inf.ed.ac.uk/infweb/research/study-updates>.

Alternative formats.

To request this document in an alternative format, such as large print or on coloured paper, please contact Alexander Wasey - s1711767@ed.ac.uk .

General information.

For general information about how we use your data, go to: edin.ac/privacy-research

Consent

By proceeding with the study, I agree to all of the following statements:

- I have read and understood the above information.
- I understand that my participation is voluntary, and I can withdraw at any time.
- I consent to my anonymised data being used in academic publications and presentations.
- I allow my data to be used in future ethically approved research.

