

Bringing The Stupid Computer To Life

Alexander Wasey



Minf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2021

Abstract

This report gives a view of my current progress in creating an implementation of the “stupid computer”. This is a technique taught to students taking Informatics-1A at the University of Edinburgh, which seeks to help them understand the structure and execution of Haskell functions. So given the expression `sum [1, 2, 3]` a student would produce a trace such as:

```
sum [1,2,3]
= 1 + sum [2,3]
= 1 + 2 + sum [3]
= 1 + 2 + 3 + sum []
= 1 + 2 + 3 + 0
= 1 + 2 + 3
= 1 + 5
= 6
```

This project seeks to implement a program which can carry out this process.

The produced program is capable of carrying out this process for expressions that contain function calls, list comprehensions, and if-then-else statements. Functions called can include guards and pattern matching. Implementing the stupid computer required implementation of a formal-actual mapping system, and a method for navigating patterns and guards, among others.

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.

My gratitude also to my family, without whom this would have not been possible.

Finally I must thank my flatmates Maya, Aaron, Jack and Aidan, I couldn't hope for finer friends.

Table of Contents

1	Introduction	1
1.1	The stupid computer	1
1.2	Summary of results	2
1.3	Report summary	2
2	Existing Projects	3
2.1	Lambda Bubble Pop	3
2.2	DrRacket	4
3	Design	6
3.1	Overview of the stupid computer	6
3.2	Reduction strategy	7
3.3	How reductions work	8
3.3.1	User defined functions	8
3.3.2	If-then-else	11
3.3.3	List Comprehensions	12
3.3.4	Lists and Tuples	13
3.3.5	Undefined functionality	13
4	Implementation	14
4.1	Approach overview	14
4.2	Parsing	14
4.3	Searching for a redex	16
4.4	Reducing a redex	17
4.4.1	HsVar	17
4.4.2	HsApp or OpApp	17
4.4.3	HsIf	20
4.4.4	HsDo	20
4.4.5	Other node types	22
4.5	Printing the AST	22
4.6	Function body identification	23
4.7	Formal-Actual mapping	24
4.8	Evaluating with GHC	27
4.9	User Interface	28
4.9.1	Specifying which functions will be fully reduced	29
4.9.2	Specifying which definitions functions have	30

4.9.3	Communicating errors	30
4.10	Application distribution	31
5	Evaluation	32
5.1	Forced evaluation of arguments	33
5.2	Loss of AST structure	33
5.3	Type variable issues	34
5.4	Pattern matching against functions	34
5.5	Incorrect operator precedence	35
5.6	Support for patterns in comprehensions	35
5.7	Lack of feature support	36
6	Conclusions	37
6.1	Future developments	37
6.1.1	Representing call-by-name	37
6.1.2	Representing Sharing	38
6.1.3	Improvements to GHC interpretation	39
6.1.4	Fixing operator precedence	39
6.1.5	User interface improvements	40
6.1.6	User feedback	40
	Bibliography	41

Chapter 1

Introduction

1.1 The stupid computer

Haskell [10] is the first programming language taught to Informatics students at the University of Edinburgh, namely within the course Informatics-1A: Functional Programming [6]. The stupid computer is a concept used in this course to help students understand the structure and execution of Haskell functions. Say for example a student wishes to understand how the `foldr` function operates, the first thing to try may be looking at the function definition, which is as follows.

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

For a beginner, this may be insufficient to convey what the function does and how it does it.

Instead the stupid computer has the student evaluate the expression themselves, doing each step by hand. For example, using `foldr` to sum a list works as follows.

```
foldr (+) 0 [1, 2, 3]
= 1 + (foldr (+) 0 [2,3])
= 1 + (2 + (foldr (+) 0 [3]))
= 1 + (2 + (3 + (foldr (+) 0 [])))
= 1 + (2 + (3 + 0))
= 1 + (2 + 3)
= 1 + 5
= 6
```

The technique is used extensively in the course, and is included within the course textbook "Introduction to Computation" [14]. The technique is essentially an execution trace, though one intended to help with understanding the program, not debugging it. This trace is presented within the context of Haskell, which can be useful for those not familiar with the language. The goal of this project is to create a program to automate the stupid computer process.

1.2 Summary of results

The program produced is able to produce traces for user-defined functions and list comprehensions. These functions are able to make use of pattern matching, guards, and if-then-else statements. Partial support is also included for lambda expressions and arithmetic sequences. Users also have full access to functions from the Haskell prelude, and as such they can be used within their functions and comprehensions.

This has been achieved by interfacing with the Glasgow Haskell Compiler (GHC) API [15]. The lexer and parser it provides have been leveraged to produce Haskell parse trees, the manipulation of these trees makes up the core of the programs implementation. Hint [1] provides access to a Haskell interpreter, which is utilised to allow users access to functions from the Haskell prelude [19]. This functionality is also used to implement the formal-actual mapping and function body identification components of the program.

The program still has a number of shortcomings which I hope to address within the second year of the project. The main shortcomings are a lack of support for representing Haskell's lazy evaluation and sharing.

1.3 Report summary

Chapter 2 contains a summary of existing projects with somewhat similar aims. Chapter 3 sets out the design of the stupid computer, describing how various parts of Haskell syntax are reduced. Chapter 4 sets out the implementation of the program. This includes the user interface, distribution to users, and the implementation of the underlying stupid computer reduction system. Implementation of this underlying system required the parsing of Haskell source code, implementing a formal-actual mapping system for pattern matching, and a system for determining a functions definition based on its inputs. Chapter 5 gives an overview of my testing of the program, against a range of code samples mainly from Informatics-1A. Finally chapter 6 includes an overview of future developments to the program.

Chapter 2

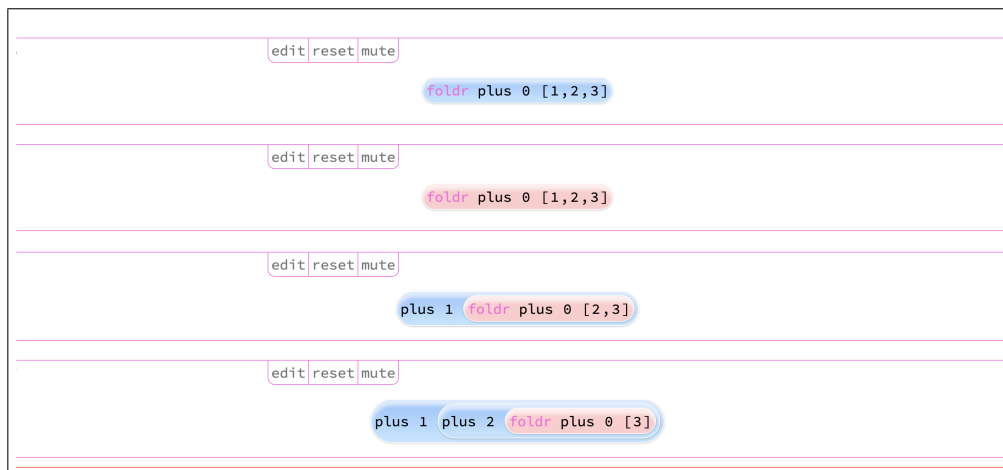
Existing Projects

The first step on such an endeavour is to examine projects with similar functionality or aims, and evaluate if they are suitable for adaptation to our needs.

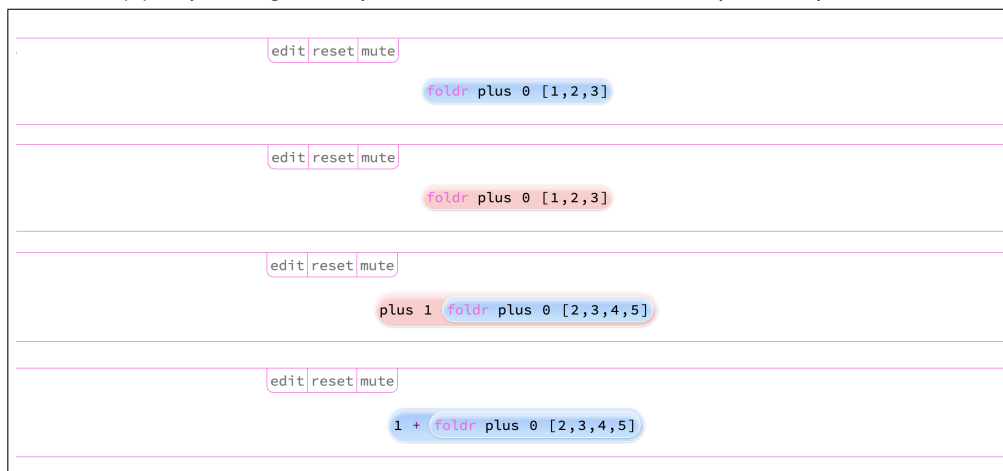
2.1 Lambda Bubble Pop

Lambda Bubble Pop [21] aids similar aims. While it has been designed as a tool for understanding the Lambda Calculus it does use a subset of Haskell, so may be suitable. Notably it is delivered via a web page, making it easy to access for students. It allows the user to select which redex to reduce on each step. This gives users the ability to reduce the expression in any number of ways, for example in the order of the Stupid Computer (Figure 2.1a) or another order entirely (Figure 2.1b). The expressions in each figure appear sequentially, not at the same time. Figure 2.1c shows the input box available to users such that they can define their own functions.

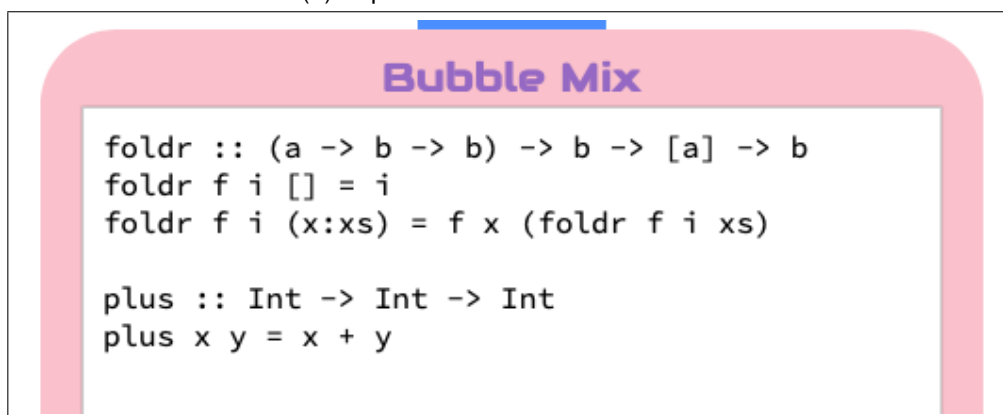
The subset of Haskell available is however limited, with no support for features such as strings or multiplication. While it could be expanded to bring in more of the language, this would be a laborious process. It is also implemented in JavaScript, which will complicate any use of existing Haskell libraries and tools. One such tool would be the Glasgow Haskell Compiler (GHC) [15], which gives access to a lexer and parser for Haskell, not having access to it will necessitate reimplementing this functionality. This would be time consuming, and therefore will be avoided.



(a) Expanding the expression in the order of the Stupid Computer.



(b) Expansion in another direction.



(c) Input box that allows users to define functions.

Figure 2.1: Lambda bubble pop.

2.2 DrRacket

DrRacket [9] includes an “algebraic stepper” for the Racket language [8]. The stepper window has two panes, the left contains the expression before reduction, the right

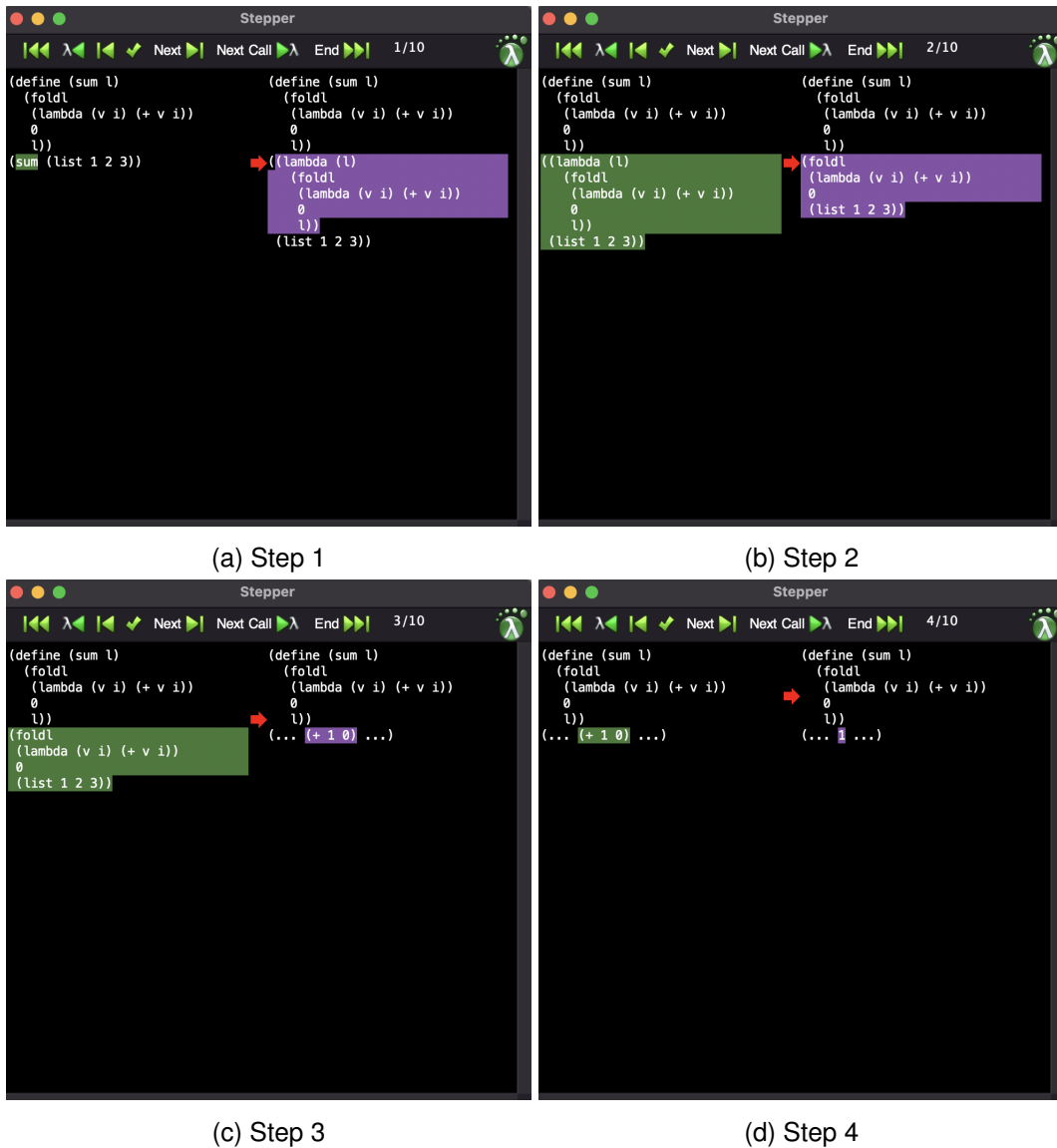


Figure 2.2: First 4 steps of evaluating the sum function written in Racket

shows the expression after reduction. The redex selected for reduction is highlighted in the left pane, and its reduced form highlighted in the right. Figure 2.2 shows the first few steps of an evaluation of the `sum` function. Highlighting the current redex is useful, as is the two pane approach. A drawback is the fact only one step of the reduction can be seen at once, it would be useful to be able to see the entire process at once. Also as seen in steps three and four often DrRacket will often not show the entire expression to the user, this seems to negate the benefit of highlighting the next to be reduced expression. Overall DrRacket is close to what we hope to achieve with the stupid computer, just for another language!

Chapter 3

Design

In this chapter an overview of the design of the stupid computer, and the rationale behind it, is given.

3.1 Overview of the stupid computer

The stupid computer process will when given the definition of a function (such as this definition for `sum`).

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

Alongside an expression, such as:

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

Will result in the following series of Haskell expressions:

```
= 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
```

The first line is the expression the user wishes to examine (in this case `sum [1,2,3,4]`). The change from one line to the next represents a single reduction step. At each step a redex is selected and replaced with its reduced form. For example `sum [2,3,4]` is the selected redex on the second line, and is replaced with `2 + sum [3,4]` on the third line. This follows from the last line of the definition of `sum`

```
sum (x:xs) = x + sum xs
```

taking $x = 2$ and $xs = [3, 4]$. The selected redex on line three is $\text{sum } [3, 4]$, in line four it is $\text{sum } [4]$ and so on. This process finishes when no more reductions are possible, and (hopefully!) the last line will be the result of the initial expression.

The rest of this chapter describes the reduction strategy, and how reduction works for each type of redex.

3.2 Reduction strategy

The reduction strategy determines which redex will be reduced at a given step. The stupid computer's strategy generally attempts to reduce the leftmost innermost redex at each step, in what is essentially a call-by-value strategy. This means a functions arguments must be reduced before the function itself. So for example given the function `add` as follows:

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

If the expression at the current reduction step is:

```
add (2*3) (4*5)
```

then the leftmost innermost redex is $(2*3)$, which is reduced to 6 giving the following:

```
= add 6 (4*5)
```

The next redex is $(4*5)$ which is reduced to 20.

```
= add 6 20
```

At this point `add` can finally be reduced giving the final two steps of the reduction.

```
= 6 + 20
```

```
= 26
```

Notably this approach differs from Haskell's call-by-name approach, where function applications are evaluated before their arguments. Following Haskell's strategy for `add (2*3) (4*5)` would result in the following reduction.

```
add (2*3) (4*5)
= (2*3) + (4*5)
= 6 + (4*5)
= 6 + 20
= 26
```

Here `add` is reduced before its arguments, unlike in call-by-value. Unfortunately it has not been possible to implement a call-by-name strategy for this year of the project. This is due to difficulties implementing a system to deal with pattern matching against function arguments. The current system requires the full evaluation of arguments before they can be matched (see Section 4.7), but I have plans to remedy this in the second year of the project (see Section 6.1.1). Sharing [11] is also not represented at

this stage of the project, but I plan to do so in the next year of the project (see Section 6.1.2).

There are two situations which call for deviation from the leftmost innermost strategy. Firstly in the `sum` example seen previously (Section 3.1) some apparently reducible redexes were passed over, such as in line 2 (`1 + 2 + sum [3, 4]`) the redex `1 + 2` is ignored in favour of `sum [3, 4]`. This is done because reducing `1 + 2` at this stage would misrepresent the nature of Haskell's recursion, due to the way the function calls have occurred line 2 is somewhat equivalent to `1 + (2 + sum [3, 4])`.

The other situation in which the strategy differs are if-then-else statements ([10], Chapter 3.6), as errors are very likely if arguments are reduced before the if-then-else, more details about this are given in section 3.3.2.

3.3 How reductions work

This section outlines how different types of expressions are reduced. These methods were decided taking the following goals into account.

- At each step the output should be valid Haskell, such that the user could, if they wished, execute it with the Haskell compiler (GHC).
- Identifiers should also be kept the same, such that users can relate the output easily back to their own code.
- 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.
- Only show the evaluation of code that the user wishes to examine. This is to ensure that the output stays useful, and doesn't overload the user with information.

3.3.1 User defined functions

Function applications (where the function is user-defined) are expanded. This involves replacing the application with the functions body, where the formal arguments within it have been replaced by the actual arguments from the application. So for example say the user wishes to examine `pyth`, which is defined as:

```
pyth :: Int -> Int -> Int
pyth x y = (x*x) + (y*y)
```

And their starting expression is `pyth 3 4`, then the reduction of the function application will be as such:

```
= pyth 3 4
= (3*3) + (4*4)
```

Here the formal variable `x` has been replaced with the actual value of 3, and the formal `y` with 4.

3.3.1.1 Pattern Matching

Pattern matching ([10], Chapter 3.17) complicates expansion in two ways. Firstly it means a single function may have multiple bodies it can be expanded to. Secondly it requires a more complex mapping between formal variables and actual parameters.

So for example with the following definition of `sum` the `(x:xs)` means the body `x + sum xs` will be chosen if the input is a non-empty list, and `[]` means the body `0` will be chosen if the input list is empty. In the first definition `(x:xs)` means the head of the list will be bound to `x`, and the tail to `xs`.

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

If pattern matching is encountered the stupid computer simply substitutes in the definition corresponding to the pattern which matches the inputs. This process is not shown to the user in any explicit way, so given the first case (with the argument `[1, 2, 3, 4, 5]`) the output will be as such:

```
= sum [1, 2, 3, 4, 5]
= 1 + sum [2, 3, 4, 5]
```

And in the second case (with the argument `[]`):

```
= sum []
= 0
```

3.3.1.2 Guards

Guards ([10], Chapter 4.4.3) are a further way function bodies can be selected, after pattern matching has occurred. Guards associate a single pattern with multiple function bodies, of these each is associated with a condition expression, these conditions are evaluated in turn and the first that evaluates to a `True` value is selected. In the following definition of `sum` the first condition evaluated is `not (null xs)`, if the result is `True` then the body `head xs + sum (tail xs)` is selected. If the result was `False` then the next condition to be evaluated will be `otherwise`, which in guards always evaluates to `True`, and therefore the body `0` will be selected.

```
sum xs | not (null xs) = head xs + sum (tail xs)
      | otherwise = 0
```

Representation of guards takes a little more thought than pattern matching, initially it seems they could be dealt with as if they were if-then-else statements (see Section 3.3.2), but alas this is not the case. I will show this by now reducing `sum` with this strategy.

```
sum [1, 2, 3]
= | not (null [1, 2, 3]) = head [1, 2, 3] + sum (tail [1, 2, 3])
  | otherwise = 0
= | not (False) = head [1, 2, 3] + sum (tail [1, 2, 3])
```

```

    | otherwise = 0
= | True = head [1, 2, 3] + sum (tail [1, 2, 3])
  | otherwise = 0
= head [1, 2, 3] + sum (tail [1, 2, 3])
= 1 + sum (tail [1,2,3])
= 1 + sum ([2, 3])
= 1 + | not (null [2, 3]) = head [2, 3] + sum (tail [2, 3])
      | otherwise = 0
...

```

Unfortunately nearly every line here is not valid Haskell. Guards should only appear as part of a function definition, not within standalone Haskell expressions. Therefore this method is not suitable, I have considered four alternatives.

1. Skip directly to the appropriate definition, similar to how pattern matching is dealt with. This would result in a loss of detail, but the output will be valid Haskell.
2. Replace the guards with equivalent if-then-else statements. This would keep it as valid Haskell, but changing syntax could be confusing for users.
3. Replace the guards with multi-way if [17]. The `sum` example would be as follows after the replacement:

```

sum :: Num a => [a] -> a
sum xs = if | not (null xs) -> head xs + sum (tail xs)
           | otherwise -> 0

```

So in this case the first few reduction steps of `sum [1, 2, 3]` would be:

```

sum [1,2,3]
= if | not (null [1,2,3]) -> head [1,2,3] + sum (tail [1,2,3])
    | otherwise -> 0
= if | not False -> head [1,2,3] + sum (tail [1,2,3])
    | otherwise -> 0
= if | True -> head [1,2,3] + sum (tail [1,2,3])
    | otherwise -> 0
= head [1,2,3] + sum (tail [1,2,3])
= 1 + sum (tail [1,2,3])
= 1 + sum [2,3]
= 1 + if | not (null [2,3]) -> head [2,3] + sum (tail [2,3])
        | otherwise -> 0
...

```

This would be less confusing and show the full reduction of the conditions, however multi-way if syntax is not part of standard Haskell and only available as a GHC extension. This option would therefore not be particularly in the spirit of keeping the output valid Haskell.

4. Output an error message telling the user guards aren't supported.

Not supporting guards (4) would be a major shortcoming as guards are a widely used in Informatics-1A, so supporting them is preferable. Solutions two and three both involve modifying the users code, and three involves non-standard Haskell syntax. With these considerations I opted for solution one, while it doesn't show the detail of reducing the guard conditions, it avoids modifying the users code, or using non-standard syntax.

The first few steps of the reduction of `sum` are therefore as follows:

```
sum [1, 2, 3]
= head [1, 2, 3] + sum (tail [1, 2, 3])
= 1 + sum (tail [1, 2, 3])
= 1 + sum [2,3]
= 1 + head [2,3] + sum (tail [2,3])
= 1 + 2 + sum (tail [2,3])
...
```

3.3.2 If-then-else

If-then-else statements ([10], Chapter 3.6) are essentially syntactic sugar for the following function:

```
if :: Bool -> a -> a -> a
if True x _ = x
if False _ y = y
```

If-then-else statements could therefore be simply treated as a function, however this would present an issue. Say for example a user defined `sum` as follows:

```
sum :: [Int] -> Int
sum xs = if (not (null xs)) then head xs + sum (tail xs) else 0
```

If the user wanted to reduce `sum []` this would involve reducing `head []`, which as `head` is only defined for non-empty lists is impossible. To avoid this issue the leftmost innermost strategy is departed from, only the condition argument will be reduced before the if-then-else statement. Therefore the trace of `sum [1,2,3]` begins as follows.

```
sum [1, 2, 3]
= if (not (null [1, 2, 3]))
  then head [1, 2, 3] + sum (tail [1, 2, 3])
  else 0
= if (not (False))
  then head [1, 2, 3] + sum (tail [1, 2, 3])
  else 0
= if (True)
  then head [1, 2, 3] + sum (tail [1, 2, 3])
  else 0
= head [1, 2, 3] + sum (tail [1, 2, 3])
...
```


3.3.3 List Comprehensions

List comprehensions [20] are widely used in Informatics-1A, so it is desirable to be able to include them in traces. There are three types of components in a Haskell list comprehension - generators, guards, and a body. Using the comprehension

```
[x*x | x <- [1,2], x > 1]
```

as an example, in this case $x*x$ is the body - this expression produces each element in the resulting list. Generators, in this case $x <- [1,2]$, feed each value from their list into the body. Multiple generators can be used, in which case all combinations of values from their lists are fed into the body. Each generated body expression is only included in the resulting list if it satisfies all of the guard expressions, in this case $x > 1$ is a guard. The resulting list of expressions therefore would be $[2*2]$, and the actual result $[4]$.

At each reduction step the first generator or guard on the right hand side the comprehension is used to determine the appropriate reduction for the comprehension.

- If the first expression on the right hand side is a generator then create two new list comprehensions, such that:
 - The first is the original comprehension, but with the variable being generated replaced by the first item in the generator, and that generator removed.
 - The second is the original comprehension but the generator has been replaced with its tail. Note that if the tail is empty then this comprehension shouldn't be included.

These lists are then concatenated together in place of the original comprehension. As an example:

```
[x*x | x <- [1,2], x > 1]
[1*1 | 1 > 1] ++ [x*x | x <- [2], x > 1]
```

- Otherwise the first expression on the right hand side will be a guard. If possible the guard will undergo a reduction step. Once no further reductions of the guard are possible, then there are two possibilities:
 - If the guard reduced to `True` then remove the guard from the comprehension.
 - Otherwise it was reduced to `False`. In this case the comprehension is replaced with an empty list.

An example of both cases follows:

```
[2*2 | 2 > 1]
[2*2 | True]
[2*2]
```

```
[1*1 | 1 > 1]
[1*1 | False]
```

```
[ ]
```

Once the comprehension has no generators or guards remaining it is converted to a normal Haskell list, which contains just the body expression.

So a full reduction of this list 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]
```

3.3.4 Lists and Tuples

As may be expected, the redexes inside a list or tuple are reduced left to right, each being fully reduced before moving onto the next. So for example this looks like:

```
= [1*1, 2*2, 3*3]
= [1, 2*2, 3*3]
= [1, 4, 3*3]
= [1, 4, 9]
```

3.3.5 Undefined functionality

In the case of a redex that has no defined reduction it is immediately reduced to normal form. The primary example of this are function applications which make use of functions which are not user-defined, but instead come from the Haskell prelude [19]. For example in the following “*” comes from the prelude, not a user definition, and therefore is immediately reduced to normal form.

```
3 * 3
= 9
```

This also occurs for Haskell syntax that does not have a defined reduction strategy, for example lambda expressions ([10], Chapter 3.3). In the following the application of the lambda expression is immediately reduced to normal form.

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

This is implemented by evaluating the undefined redex with the GHC interpreter provided by `hint [1]` (also see Section 4.8). This gives users the flexibility to use a wide range of Haskell functions and syntax, without them needing to be explicitly accounted for.

Chapter 4

Implementation

This chapter covers the current implementation of the stupid computer. Haskell has been used for the implementation, as it gives access to a wide range of existing tools for working with Haskell source code.

4.1 Approach overview

My approach is based on manipulations of abstract syntax trees. The users main expression is parsed into such a tree, and each reduction step is implemented as a manipulation of this tree. Below are the first three steps of reducing `sum [1, 2, 3, 4]` and the corresponding series of ASTs (see Figure 4.1).

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

At each of these reduction steps the following needs to occur:

1. The redex must be identified, within the AST this means finding the root node of the subtree that represents it.
2. The AST of the reduced redex is constructed.
3. This AST is substituted into the main tree in the place of the redex root node.
4. The updated main AST is then printed back to Haskell source code, and shown to the user.

This process repeats until no more redexes exist within the AST. The following sections lay out the full detail of each part of this implementation.

4.2 Parsing

Before any of the reduction steps can occur the users input must be parsed into an AST. I considered using the Alex [7] lexer generator, and the Happy [12] parser generator

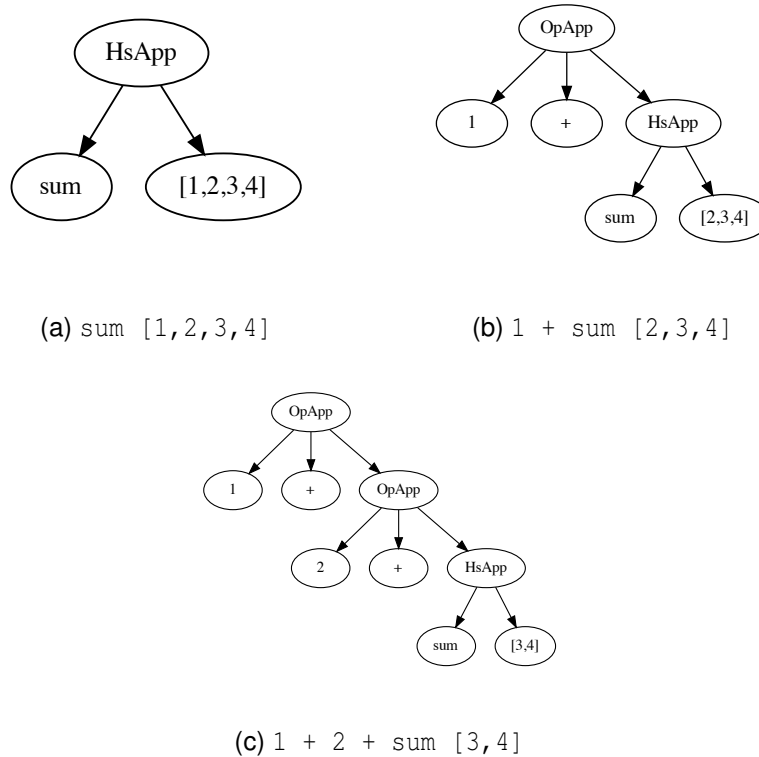


Figure 4.1: Series of (simplified) ASTs, each representing a reduction step.

along with my own grammar to implement the parsing. However I instead decided to use the lexer and parser belonging to GHC, which conveniently makes these tools available as a library via the GHC API [15]. After the parsing stage the user-defined functions are identified. For each a record is created, containing the ASTs for its type signature and function definition, alongside its identifier, and the number of arguments it requires. The AST [16] which represents the main expression is also extracted, for reduction by the rest of the program.

Using GHC has several benefits over implementing my own solution.

- Eliminates the development time of developing a lexer and parser.
- As Haskell is updated, the GHC API will reflect those changes, this will make it easier to keep the stupid computer up to date with new Haskell versions.
- It has full coverage of the Haskell specification.
- The GHC API has functionality for pretty-printing GHC ASTs back to Haskell source code.

Though there are a few drawbacks.

- The produced AST has more complexity than the stupid computer needs.
- It means the stupid computer is constrained by the rules of the GHC AST. It could be useful to have the AST represent non-valid Haskell, but as I am using

the GHC AST this is often not possible.

- The parser parses all infix operations as left-associative, meaning the expression $2+3*4$ is parsed as $(2+3)*4$ rather than $2+(3*4)$. Only after parsing does GHC re-associate the infix operations correctly, using the renamer [13]. The main effect of the renamer however is fully resolving all qualifiers, so for example `sum` becomes `Data.Foldable.sum`, and `+` becomes `GHC.Num.+`. Using the renamer would therefore require undoing this change. As I only discovered this left-associative parsing issue during my evaluation of the system I have not yet been able to implement a solution, whether using the renamer or otherwise. Details of possible solutions can be found in section 6.1.4.

4.3 Searching for a redex

The first step of each reduction is identifying the redex to be reduced, and the node within the AST at the root of its subtree. This is done by traversing the tree in dept-first post-order searching for such nodes. Table 4.1 is a summary of the types of nodes of interest, and which kinds of redexes they may represent.

Node	Redex
HsVar	Function or constructor
HsApp	Prefix function application
OpApp	Infix function application
HsIf	If-then-else
HsDo	List comprehension

Table 4.1: A summary of node types of interest, and the redexes they may represent.

When a node is traversed not only is a new node returned, so is a `TraverseResult`. This is a bespoke data type used to communicate the result of the traversal, this is needed to ensure only one reduction happens in each reduction step. For example if a parent traverses one of its child nodes and the `TraverseResult` is marked as **Reduced** then some redex within that subtree has been reduced. As only one reduction can occur at each reduction step the parent knows not to traverse any of its other children or reduce itself. There are four possible values of `TraverseResult`.

1. **Reduced** - This indicated that a regex within the subtree has been reduced. This indicates to the parent nodes that they should not attempt any reduction until the next reduction step.
2. **Found** `args name` - Indicates a user-defined function has been located, but does not yet have enough arguments to be reduced. `args` indicates the number of arguments already found in the subtree, and `name` is simply the name of the function. When enough arguments have been found this function application can be reduced.
3. **NotFound** - Indicates no reducible regex has been found in the subtree. In this

case the stupid computer may attempt to reduce the entire subtree via GHC evaluation.

4. **Constructor** - Returned when the subtree represents the application of a constructor, in this case no GHC evaluation should be attempted.

When the traverse of the main expression root node returns any value other than a **Reduced** then no further redexes exist in the main expression, and the reduction halts.

4.4 Reducing a redex

Once a node at the root of a possible redex has been found two conditions must be fulfilled before it can be reduced.

- The subtree does in fact represent a redex.
- The redex is next in line for reduction.

The rest of this section details the checking of these conditions, how the reduction occurs, and the `TraverseResult` values in each case.

4.4.1 HsVar

This node represents the name of a function or constructor. There are three cases for what the `TraverseResult` could be.

- If the name begins with an uppercase letter than the `HsVar` represents a constructor. In this case **Constructor** is returned, so no reduction of the constructor application is attempted.
- If the name is that of a user defined function then the number of arguments needed by the function is checked.
 - If zero arguments are required by the function (for example if it is a constant), then the `HsVar` is replaced with the `HsExpr` representing the functions body. **Replaced** is the traverse result in this case.
 - Otherwise more arguments are required, the traverse result is (**Found** 0 `FunctionName`)
- Finally if the function is not user defined, an **NotFound** traverse result is returned.

4.4.2 HsApp or OpApp

Both of these nodes represent a function application. As a call-by-value strategy is being followed their arguments must be fully reduced before the application.

First the nature of their operator must be determined, to ensure it is fully reduced, and to determine if it is user-defined. This is done by traversing the operator, for an `HsApp` this is its left-hand side child, for an `OpApp` its middle child. In either case if

the traverse returned a **Reduced** then their operator was not fully reduced, and so the application cannot be reduced on this iteration. Otherwise the other children of the node are traversed, for an `HsApp` just the right-hand side child, for a `OpApp` both the left and right child. If any of those traversals returned a **Reduced** then a reduction of the application is still not possible on this iteration.

If no reduction has yet occurred then based on the result of the traversal of the operator there are three possibilities.

4.4.2.1 Constructor

As this is a constructor application no reduction is possible, therefore the **Constructor** value is returned back up the tree.

4.4.2.2 (Found args name)

This indicates that the operator contains a user-defined function. The number of arguments within the operator is stored within `args`, and the name of the user-defined function within `name`.

Reduction of the application can only occur once the function has enough arguments. If `args+appArgs == argsNeeded` then the reduction can proceed. (`appArgs = 1` iff at a `HsApp` node, 2 otherwise.) Otherwise more arguments are needed, this is communicated by returning `(Found (args+appArgs) name)` up the tree.

If the function can be reduced then the application node must be replaced with the body of the function being called, with its formal variables substituted for the appropriate actual parameters. For example the following reduction step is implemented with the AST manipulation in figure 4.2.

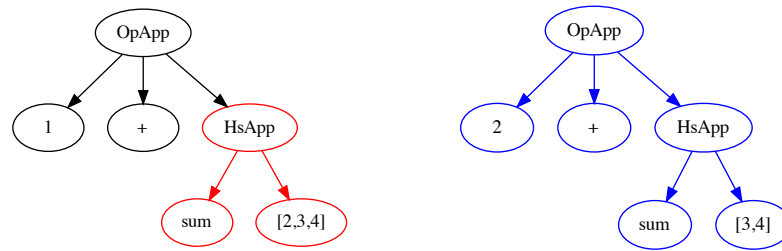
```
= 1 + sum [2,3,4]
= 1 + 2 + sum [3,4]
```

To generate the AST which represents the reduced application takes four steps:

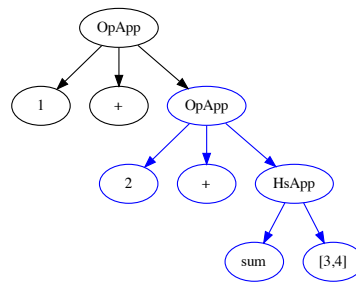
1. Identify the function arguments, this is achieved with a depth first search of the function application subtree.
2. Identify the appropriate function body for substitution. As seen previously a single Haskell function can have many possible bodies to substitute into the AST, depending on the outcomes of pattern matching and guards. The following function has three such possible bodies.

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

The stupid computer must determine which of these will be substituted into the AST, section 4.6 describes in detail how this is achieved.



(a) Main AST, the red subtree is the function call to be reduced. (b) AST representing the reduction of `sum [2, 3, 4]`.



(c) AST after reduction.

Figure 4.2: Single step of reduction, as implemented as a manipulation (of a much simplified) AST.

- Acquire a mapping between the functions formal arguments, and its actual parameters. For example the function call `gt 0 [5, 10, 15]` (using the definition of `gt` above) results in the formal-actual map seen in Table 4.2.

Formal	Actual
x	0
y	5
ys	[10, 15]

Table 4.2: Formal-Actual Map for `gt 0 [5, 10, 15]`.

The system to achieve this mapping is detailed in section 4.7.

- Substitute the formals in the function body AST for the actual values. This is achieved by traversing the AST, and replacing any encountered formal variables with the actual value that has been mapped to it.

The HsApp or OpApp node is then replaced with this new AST, and a **Replaced** value of `TraverseResult` communicated up the tree.

4.4.2.3 NotFound

A **NotFound** traverse result indicates that the operator contains either a function that is not user-defined, or some syntax that does not have a defined reduction strategy (such as lambda expressions). Therefore an attempt is made to reduce this application directly to normal form, this is done by trying to evaluate the application using GHC (see Section 4.8). If this fails then the application does not yet have enough arguments, and the same **NotFound** traverse result is propagated up the tree so more can be found. If the execution succeeds then the result is placed within a `HsVar` in place of the application node. This gives the user the flexibility to use all functions from the Prelude [19], but much like the formal-actual mapping system the result loses any structure within the AST (see Section 4.7).

4.4.3 HsIf

If-then-else expressions are represented by a `HsIf` node. As seen in section 3.3.2 only the condition must be fully reduced before the `HsIf` itself. Therefore first the condition is traversed, if the result is **Reduced** then the condition was not yet fully reduced, therefore the `HsIf` cannot be reduced at this step. Otherwise no reduction occurred in the condition, and reduction of the if-then-else is possible. If the condition is equal to "True" then the expression is reduced to its True-value, if equal to "False" it is reduced to its False-value. This makes the assumption the condition reduces to either "True" or "False" - this will always be the case as the condition must have a `Bool` type. A **Reduced** traverse result is then returned up the tree.

4.4.4 HsDo

`HsDo` nodes which have their `HsStmtContext` field set to `ListComp` represent a list comprehension within the AST. If it does represent a comprehension then the components of the comprehension are stored in a list of `ExprLStmt` expressions. Each contains a `LStmt`, which each contain a `StmtLR`. Which type of `StmtLR` represents each part of a comprehension is summarised below.

- Generators - `BindStmt`, this contains both the source of the values, and the pattern they will be matched against.
- Guards - `BodyStmt`
- Body - `LastStmt`, the GHC parser always places this as the end of the list.

So for example the following list comprehension;

```
[(x,y) | 1<2, x<-[1,2,3], odd x, y<-[1,2,3], x<y ]
```

will be represented by the following list of `ExprLStmt` expressions:

```
[BodyStmt{1<2}, BindStmt{x<-[1,2,3]},
  BodyStmt{odd x}, BindStmt{y<-[1,2,3]},
  BodyStmt{x<y}, LastStmt{(x,y)}]
```

The nature the reduction depends on which kind of `ExprLStmt` is at the head of the list.

- `BindStmt` (**Generator**). The `HsExpr` that represents the list within the bind is traversed, following this there are two cases.
 - If the result of the traverse was a **Reduced** then no reduction happens to the comprehension, and the **Reduced** is returned up the tree.
 - Otherwise the generator was already fully reduced, in this case the assumption is made that the generator list is represented as an `ExplicitList`. This list is used to create the two new list comprehensions.
 - * In the first the identifier being bound is substituted for the head of the generator list and the generator removed. If only a `LastStmt` (Body expression) remains then the comprehension is converted to a `HsExplicitList` containing just the `HsExpr` within the `LastStmt`.
 - * The other is essentially identical, but with the tail of the generator replacing the original.

This assumption about the list being in the form of an `ExplicitList` does present a limitation, if the list is a result of a GHC interpretation it will have been squashed to a `HsVar`. If the list is not in the form of an `ExplicitList` then the comprehension cannot be properly reduced, and therefore will be evaluated with GHC (see Section 4.8). The solution for this issue will be to keep the proper structure of the AST throughout reduction, details of this are specified in sections 6.1.1 and 6.1.3. Additionally the stupid computer is currently lacking support for `BindStmt` nodes which make use of pattern matching, for example in the following comprehension:

```
[x | (x,y) <- [( 'H' , 0), ( 'e' , 1) ]]
```

The formals within the pattern will not be substituted for the actuals within the list, causing an improper reduction. This is further discussed in section 5.6.

- `BodyStmt` (**Guard**). The `HsExpr` within which represents the condition is traversed, following this there are two cases.
 - If the result of the traverse was a **Reduced** then no reduction happens to the rest of the comprehension, and the **Reduced** is returned up the tree.
 - Otherwise the guard has been fully reduced already, to either a “True” or “False” value. So in each of the two cases:
 - * “True” - The list comprehension is retained with the guard removed. Again if no generators or guards remain then the comprehension is converted to an `HsExplicitList` containing the `LastStmt`.
 - * “False” - The comprehension is replaced with an empty `HsExplicitList`.

This does make the assumption the result of the guards reduction is either “True” or “False”, which will always be the case as guards must always have a `Bool` type. In both cases a **Reduced** result is returned up the tree.

- `LastStmt` (Body expression). This won’t be encountered as the head of the list, as in the previous two cases whenever only a `LastStmt` remains in the comprehension it is converted to an `HsExplicitList`.

4.4.5 Other node types

As mentioned in section 3.3.5 when a node is encountered with no specific reduction strategy then an attempt is made to reduce it to normal form. This is done by attempting to evaluate the node using the GHC interpreter (see Section 4.8). If the evaluation succeeds then the returned value is placed within an `HsVar` which replaces the original node, and a `Reduced` result can then be returned up the tree. However a direct reduction to normal form is not always possible, for example if the node is a lambda expression which doesn’t have any provided arguments. In this case the evaluation will fail, and a `NotFound` result will be returned up the tree.

This gives users the flexibility to use Haskell syntax that has not been explicitly accounted for, such as `ArithSeq` nodes [16], which represent arithmetic sequences, for example `[1..5]` ([10], Chapter 3.10). However this does not work for all undefined statements, for example if the node contains formal variables they will not have been substituted for actual values, and therefore the evaluation will fail.

4.5 Printing the AST

The GHC API includes a pretty printer [2] for displaying the AST as Haskell source code. This works mostly as expected, though sometimes it can format the results oddly. One example of this is its auto-insertion of line breaks, often this works well and helps make the code more readable, such as below:

```
= if (not (null [1, 2, 3])) then
    (head [1, 2, 3]) + sum (tail [1, 2, 3])
  else
    0
```

In this example the line breaks work fairly well to make the code more readable by the user, and are somewhat similar to how code may actually be formatted. However in the following the line breaks are inserted somewhat strangely:

```
= []
   ++
   [8] ++ [10] ++ [12]
```

For reasons unbeknownst to me it has decided to give `[]` and `++` their own lines, what may be expected instead is for the output to all be on one line. It would be possible to resolve this by removing all newlines from the output, though I have decided against this, as I feel that having spurious line breaks is much better than having none at all.

One step that is being taken to improve readability of output is removing some unnecessary parenthesis. For example if a user writes the `map` function in the following way:

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

The output without removing parenthesis if called as `map square [1,2,3]` is:

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

The parenthesis around `f x` in the definition are around 1 and 4 in the trace. This isn't particularly pleasant to read. Of course it is not desirable to remove all parenthesis, as they are sometimes needed to show precedence, and can often make the code more readable. As a compromise only parenthesis that surround a single value, such as an integer or list, are removed. If an expression within parenthesis is reduced the result is checked to see if it is a single value, for example if it is a `HsVar` or `HsLit`. If so the parenthesis are removed. The same process happens whenever a variable is substituted for an actual value. The output will therefore be:

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

Now the parenthesis are far less overwhelming, and the output more easily parsed by the user.

4.6 Function body identification

As seen previously a single Haskell function can have many possible bodies to substitute into the AST, depending on the outcomes of pattern matching and guards. The following function has three such possible bodies.

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

To determine which of these bodies are appropriate given the arguments, a new function is defined. It is identical to the function being expanded except each of the possible bodies is replaced with its numbering. The name of the new function has been prefixed with `defgetgt` to avoid clashes with the definition of the original function. The function's type has been removed, and GHC's type inference relied upon instead. For the `gt` function above this function would be:

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

A map is also created between these numberings and the AST of the function body they represent. This function is then executed via GHC (see Section 4.8) with the same arguments as the original, the result is then used to determine the appropriate definition for substitution, such as in table 4.3.

Sample arguments	Function call	Function result	Body
10 []	<code>defgetgt 10 []</code>	0	[]
10 [15,5]	<code>defgetgt 10 [15,5]</code>	1	<code>y : gt x ys</code>
10 [5,15]	<code>defgetgt 10 [5,15]</code>	2	<code>gt x ys</code>

Table 4.3: Possible function inputs, the call evaluated by GHC, the value returned by the new function, and the relevant function bodies for substitution.

As this function is executed with GHC it gives the user the flexibility to use any patterns they wish, alongside being able to use all of standard Haskell within guards. This does mean that GHC has to be invoked every time this process takes place which is an expensive operation. While a system for dealing with the pattern matching element of this operation could have been implemented, doing an equivalent for guard conditions would have been impractical.

4.7 Formal-Actual mapping

Generating this mapping is complicated by pattern matching ([10], Chapter 3.17). If Haskell lacked pattern matching achieving this mapping would be fairly simple, it could be created using the ordering of the formals and actuals. This is not the case of course, so to deal with this another function is generated which will return a list of tuples, each containing the name of a formal and the actual value it is bound to. The AST of the pattern is traversed in order to extract the names of the formals, so for example the function generated for the `gt` example (see in Section 4.6) will be:

```
formalactualfuncgt :: Show a => Ord a => a -> [a] -> [(String, String)]
```

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

The type of the function has also been changed, the return type now being `[(String, String)]`. Another change is that all type variables must now be a member of the `Show` typeclass, this is necessary to allow all inputs to be converted to strings by `show`. This function is then executed with GHC (see Section 4.8) with the applications arguments, the list it returns is then converted to a map. This means that the actual arguments are now represented by strings, which are placed within `HsVar` nodes in order that they can be substituted into the AST.

While in this case this method works perfectly, it has a fatal flaw, what if we want to pass a function? A function cannot be a member of `Show`, so would result in a type error. This is dealt with by only including those arguments which are pattern matched in the new function, and instead adding the other arguments to the map directly. This requires a separate function to be generated for each of the functions patterns, as each could have a different number of arguments being matched against. So for the `gt` example the two new functions generated will be (for the first pattern, and the second respectively):

```
formalactualfuncgt :: [(String, String)]
formalactualfuncgt = []
```

```
formalactualfuncgt :: Show a => Ord a => [a] -> [(String, String)]
formalactualfuncgt (y:ys) = [("y", show y), ("ys", show ys)]
```

The appropriate of these functions will then be executed with GHC, with the arguments that need mapping. Which function is used depends on the outcome of function body identification as seen in section 4.6.

While this approach is able to carry out the formal-actual mapping for all of Haskell's pattern matching structures it does suffer from several issues:

1. It requires a call to GHC each time a map has to be generated. This is quite an expensive operation, however not in itself a fatal flaw, as performance is still good enough to be usable.
2. It forces all arguments to be fully evaluated when they are passed to `show`. This makes it impossible to support lazy evaluation with this system, as such all functions are call-by-value.
3. It is not possible to map infinite data structures, for example if mapping `[0..]` to `(x:xs)` the attempts to evaluate `show [1..]`. This would take infinite time and memory, and as such the reduction fails.
4. GHC type inference can fail when type variables are used, this can cause GHC to return the user a type error even though none exists in their code! For example given the following definition of `length`. and attempting to reduce `length [[]]`.

```
length :: [a] -> Int
```

```
length (x:xs) = 1 + (length xs)
length [] = 0
```

If the user wishes to reduce `length [[]]` the following function is generated:

```
formalactualfunlength :: Show a => [a] -> [(String, String)]
formalactualfunlength (x:xs) = [("x", show x), ("xs", show xs)]
```

Evaluating `formalactualfunlength [[]]` via GHC should return the formal-actual map for the `length [[]]` function application. However in this case this fails, GHC is unable to derive the type of `a`, and therefore ensure it is a member of `Show`. This failure occurs because GHC has no non-empty instances of `a`, for all it knows `a` may not actually be showable! This results in a compiler error, and the GHC evaluation fails, so no formal-actual map can be generated. This input should be valid, so therefore this is a major bug. Currently students can resolve this issue by replacing type variables with concrete types, this is communicated to them via an error message when the issue occurs (see Section 4.9.3). A potential replacement system which should resolve this issue is outlined in section 6.1.1.

5. The returned results are seen purely as strings, no longer their proper types. For example if the list `[2,3,4]` is mapped to a formal with this system it will be represented in the AST as a `HsVar` node with the name `"[2,3,4]"`. Ideally it would be stored as an `HsExplicitList` node, with a `HsLit` child representing each value. This causes issues with the implementation of list comprehensions, as they expect their generators to be in the form of a `HsExplicitList` (see Section 4.4.4).
6. The system is not capable of dealing with situations where functions are matched against, such as in the following:

```
compose :: [(a -> a)] -> a -> a
compose [] = id
compose (f:fs) = f . compose fs
```

With the initial expression:

```
compose [(map (+) 1), (map ((* 2)))] [1, 2, 3]
```

In this case the system will attempt to evaluate the following to produce the formal-actual map:

```
[("f", show (map (+) 1)), ("fs", show [(map ((* 2))])]
```

This results in a type error, and the mapping fails. This failure is communicated to the user, along with an explanation to the source of the failure (see Section

4.9.3). A potential replacement system which should resolve this issue is outlined in section 6.1.1.

A potential new system which deals with these issues is outlined in section 6.1.1.

Formal-actual mapping and definition identification could have been implemented with a single newly defined function, I decided against such a design so that the two components could be implemented and tested separately. It also allows for the replacement of one system without having to modify the other, which will be of benefit when implementing the system seen in section 6.1.1. Despite its flaws this initial system did prove sufficient for the rest of the stupid computer to be developed.

4.8 Evaluating with GHC

I have previously mentioned executing code via GHC, this is achieved using the `hint [1]` library. It makes use of the users installation of GHC to allow for interpretation of Haskell code at runtime. This gives users access to every function within the prelude. It also gives the ability to use some syntax, such as lambda expressions, which have not as of yet been properly implemented for the stupid computer.

Interpreting calls to prelude functions is very simple, the expression is simply pretty printed and passed to the interpreter. Calls to custom functions, such as when identifying function definitions, are more complex. The function in question, along with all of the user defined functions, must be provided to the interpreter. User defined functions must be included as they may have been used in guards. The interpreter only accepts a single line expression however, so all of these functions must be combined into a single string. Fortunately Haskell allows for this, indentation can be replaced by surrounding blocks in curly braces `''`, and newlines can be replaced with semicolons `';`. So each function is pretty printed, and each of its lines concatenated together, each separated with a `';`. This string is placed within curly braces, and then combined a `let` statement and the function call. I will now run through an example of how the interpreter is set up for a definition identification. Say the identification is happening for the following function:

```
sum :: Num a => [a] -> a
sum xs | not (null xs) = head xs + sum (tail xs)
       | otherwise = 0
```

Then the definition identification function is:

```
defgetsum :: Num a => [a] -> Int
defgetsum xs | not (null xs) = 0
             | otherwise = 1
```

If the argument the definition is being identified for is `[1,2,3]`, then the string passed to the interpreter would be:

```
let {sum :: Num a => [a] -> a; sum xs | not (null xs) = head xs + sum
(tail xs); | otherwise = 0; defgetsum :: Num a => [a] -> Int; defgetsum
xs | not (null xs) = 0; | otherwise = 1} in defgetsum [1,2,3]
```


Of course GHC evaluation has downsides, firstly it relies on the users install of GHC, this forces the program to be dynamically compiled, which complicates distribution. It also requires that the users entire input file be recompiled every time this system is invoked, which is fairly expensive, though the program's performance is still acceptable. Additionally each time a call is made the entire input (including the functions not used) must be compiled by GHC

Another downside is that the output of the interpreter will always be a string, and substituted into the AST within a `HsVar`. For operations such as evaluating list comprehensions this can be fatal, as my implementation of those is expecting an `HsExplicitList`. I have plans to deal with this as laid out in 6.1.3

4.9 User Interface

The user interface of the stupid computer is currently a command line interface (CLI). I have decided against implementing a graphical user interface, this would be time intensive and distract from the main purpose of the project - implementing the logic of the stupid computer. Although developing such an interface is certainly possible in the future. It is also worth noting that students already tend to interact with Haskell through the GHCi CLI [18], and as such are likely to be familiar with CLIs.

This CLI takes a single Haskell source file as input. Such files contain the definitions of every function the user wants expanded, alongside the expression they wish to see reduced. Figure 4.3 shows such an input file, which when evaluated with the stupid computer gives the output seen in figure 4.4. It must be noted that this differs from standard Haskell source files, which contain only definitions. This is inconvenient for users as it means standard Haskell source files cannot be directly used by the stupid computer. It also means that stupid computer input files cannot be directly loaded into GHCi. However this design was relatively simple to implement, and as such allowed a focus on getting the rest of the system up and running.

```

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 4.3: Example input file, here both `map` and `square` are expanded. `map square [1..3]` will be reduced.

Additionally the CLI will accept `--help` as an input, and then show usage instructions to the user, see figure 4.5.

```
alexw@Alexanders-MacBook-Pro examples % stupid-computer map.hs
  map square [1 .. 3]
= 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]
alexw@Alexanders-MacBook-Pro examples %
```

Figure 4.4: Output of the stupid computer when run on the input file shown in figure 4.3

```
alexw@Alexanders-MacBook-Pro examples % stupid-computer --help
Inputs should be given as a .hs file, such as `stupid-computer examples/sumpattern.hs`
Example inputs are available in examples/ in the source repo at:
https://github.com/alexanderwasey/stupid-computer
Such an input file may look like:

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

sum [1,2,3,4]

alexw@Alexanders-MacBook-Pro examples %
```

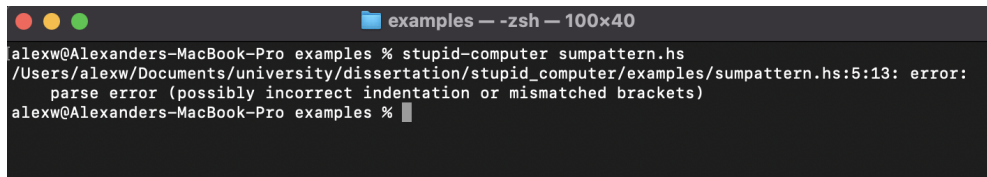
Figure 4.5: Stupid computer --help message.

4.9.1 Specifying which functions will be fully reduced

As seen previously users specify in their input file which functions have their full reduction shown, rather than being immediately collapsed to normal form. It is worth noting that other options were considered.

- Show the full reductions of only the functions in the users main expression. In this case if the user is examining the expression `map square [1,2,3]` `map` and `square` would be reduced fully. This does not give the user much flexibility with regards to which functions they want to have reduced, as all such functions would have to be in the initial expression.
- Fully reduce all functions encountered. This would deal with the previous issue, but would cause the reduction of some functions to be so detailed as to be unreadable. For example a function making use of `length` ([10], Chapter 8.1), on a list of length N would have to take $2N + 2$ reduction steps just to get that length value!

These two potential systems fail to give the user much control over what they want to see the full reductions, and as such the system wherein only functions defined by the user are fully reduced has been implemented.



```
alexw@Alexanders-MacBook-Pro examples % stupid-computer sumpattern.hs
/Users/alexw/Documents/university/dissertation/stupid_computer/examples/sumpattern.hs:5:13: error:
  parse error (possibly incorrect indentation or mismatched brackets)
alexw@Alexanders-MacBook-Pro examples %
```

Figure 4.6: Output when the users input has a syntax error, in this case `sum [1,2,3,4.`

4.9.2 Specifying which definitions functions have

Also above the implementation relies on the users input file to find the definitions of functions being fully reduced. While it would be impossible to do otherwise for user-defined functions, for those in the GHC prelude I considered instead using their definitions from the standard library. This would save a fair bit of legwork for users, they would not have to redefine every function they wanted expanded, however I decided it was not suitable due to the unexpected way many functions are defined in the prelude. For example `sum` is defined (in base-4.14.1.0 [5]) as such:

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

This definition is unexpected to say the least! For a user unfamiliar with Haskell the reduction of this will likely be of little to no use. Another issue is that the prelude definitions of standard functions change over time, for example the Haskell-98 version of `sum` ([10], Chapter 8.1) is defined as:

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

This is a far more usual way to define the function. While it would be possible to pick such a version of the prelude and stick with it, it would slowly become out of date, and could end up missing functions that a student of the future might expect to find. Ultimately given these considerations I decided upon using the users definitions.

4.9.3 Communicating errors

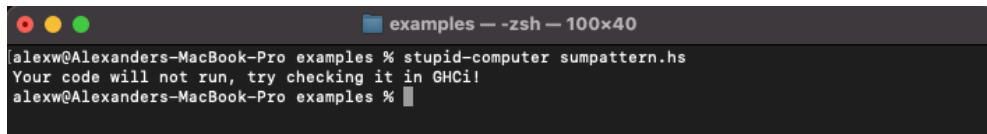
Several types of errors can occur, these must be communicated to the user. The first is a syntax error in the users input - such as mismatched brackets. The following figure 4.6 shows the output when a user gives an input file containing such an error, in this case the main expression is `sum [1,2,3,4` - which is missing a final `]`.

A users code may also be invalid because it has some logical error, for example if a user defined `sum` as:

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

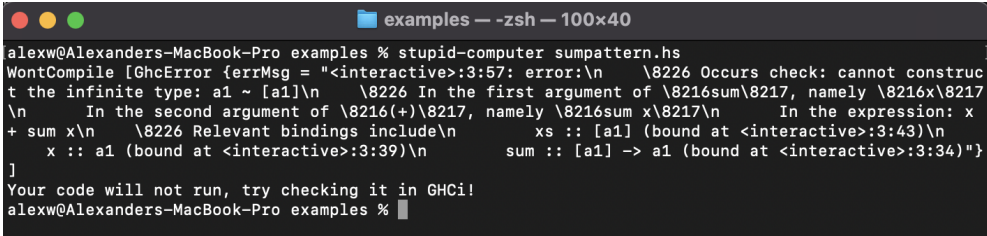
In this case there are no syntax errors, but a type error due to `sum x` in line 2 - the user is trying to sum a single number! The stupid computer output in this case is shown in figure 4.7a. Type errors are detected by the interpreter [1], but unfortunately its error

messages are not particularly easy to read (see Figure 4.7b). Therefore these messages have not currently included, and the user directed to GHCi.



```
alexw@Alexanders-MacBook-Pro examples % stupid-computer sumpattern.hs
Your code will not run, try checking it in GHCi!
alexw@Alexanders-MacBook-Pro examples %
```

(a) Output when the users input has a logical error, in this case $\text{sum } (x:xs) = x + \text{sum } x$.



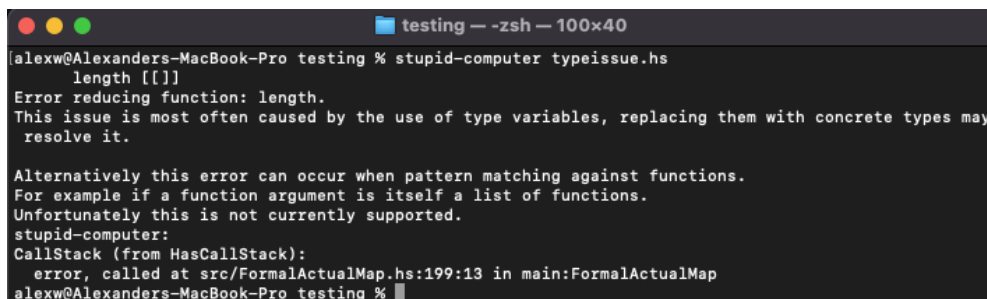
```
alexw@Alexanders-MacBook-Pro examples % stupid-computer sumpattern.hs
WontCompile [GhcError {errMsg = "<interactive>:3:57: error:\n      \8226 Occurs check: cannot construc
t the infinite type: a1 ~ [a1]\n      \8226 In the first argument of \8216sum\8217, namely \8216x\8217
\n      In the second argument of \8216(+)\8217, namely \8216sum x\8217\n      In the expression: x
+ sum x\n      \8226 Relevant bindings include\n      xs :: [a1] (bound at <interactive>:3:43)\n
      x :: a1 (bound at <interactive>:3:39)\n      sum :: [a1] -> a1 (bound at <interactive>:3:34)"}
]
Your code will not run, try checking it in GHCi!
alexw@Alexanders-MacBook-Pro examples %
```

(b) The error message given by Hint, this is not currently being used on account of its confusing formatting.

The final kind of error is one due to a failing of the stupid computer implementation. Figure 4.8 shows the output when the type variable issue or function matching issue has occurred (see Section 4.7), and suggests a course of action to the user for remedying it.

4.10 Application distribution

The distribution of the program to students must be considered. The easiest method would be to simply pre-compile executables for each platform (Windows/macOS/Linux) and make them available for students to download. Unfortunately the use of hint [1] prevents statically linking the executables; they must be dynamically linked instead. This means users must build the tool locally, and this is therefore done Stack [4]. This also allows the tool to be distributed via Hackage [3] in the future, so all students will need to do is run “stack install stupid-computer” in their terminal.



```
alexw@Alexanders-MacBook-Pro testing % stupid-computer typeissue.hs
length [[]]
Error reducing function: length.
This issue is most often caused by the use of type variables, replacing them with concrete types may
resolve it.

Alternatively this error can occur when pattern matching against functions.
For example if a function argument is itself a list of functions.
Unfortunately this is not currently supported.
stupid-computer:
CallStack (from HasCallStack):
  error, called at src/FormalActualMap.hs:199:13 in main:FormalActualMap
alexw@Alexanders-MacBook-Pro testing %
```

Figure 4.8: Output when the stupid computer encounters a type variable error, or function matching error 4.7.

Chapter 5

Evaluation

The stupid computer has been tested against a number of code samples. These have been taken from the slides and quizzes of the course Informatics-1A [6], such that they are representative of the type of code that students are likely to write. Not all samples from the course were tested against, those where a stupid computer trace has been shown or asked for have been focused on. Efforts have also been made to avoid repeating similar samples. Some of the samples have been lightly modified to remove `where` statements as they are not supported by the program at present, though one has been retained to ensure the tests do cover `where` statements. Alongside these are a lesser number of code samples written by myself. Overall there were 46 code samples, with 31 taken from Informatics-1A. Out of these 11 failed to run, and 5 ran but had unexpected (but somewhat functional) outputs. While this seems a high number of failures most come from the same few root issues, a summary of which can be seen in table 5.1.

Root cause	Tests failed	Unexpected results
Forced evaluation of arguments (Section 5.1)	lect04-5 lect04-8 lect05-1 lect05-3	
Loss of AST structure (Section 5.2)		lect04-6 lect09-02
Type variable issues (Section 5.3)	'typeissue'	
Pattern matching against functions (Section 5.4)	'compose'	
Incorrect operator precedence (Section 5.5)		'operators'
Support for patterns in comprehensions (Section 5.6)		'listpattern'
Lack of feature support (Section 5.7)	lect05-4 lect09-1 lect10a-3 lect14-1 'sumcase'	lect10a-2

Table 5.1: Summary table of testing results

In the rest of this chapter I will outline these issues and their root causes.

These tests, alongside their stupid computer trace, can be found in `tests/` in the project directory.

5.1 Forced evaluation of arguments

One source of error is the forced evaluation of function arguments, as previously discussed in section 4.7. This is a cause of failed evaluation for; `lect04-5`, `lect04-8`, `lect05-1`, and `lect05-3`.

Taking `lect04-5` as an example:

```
zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

```
zip [0..] "abc"
```

In this case the formal-actual mapping system has to try and evaluate `show [1..]`, this will of course take an infinite amount of time (and memory). Therefore the formal-actual mapping cannot be completed, and the program loops forever. Section 6.1.1 describes a new formal-actual mapping system that should deal with this issue.

5.2 Loss of AST structure

Another issue is a loss of structure in the AST, as discussed in section 4.7 and section 4.4.2.3. This causes unusual output for `lect04-6` and `lect09-02`.

The code sample and output for `lect04-6` are as follows:

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = sum [x*y | (x,y) <- zipHarsh xs ys]

zipHarsh :: [a] -> [b] -> [(a,b)]
zipHarsh [] [] = []
zipHarsh (x:xs) (y:ys) = (x,y) : zipHarsh xs ys

dot [2,3,4] [5,6,7]
  dot [2, 3, 4] [5, 6, 7]
= sum [x * y | (x, y) <- zipHarsh [2, 3, 4] [5, 6, 7]]
= sum [x * y | (x, y) <- (2, 5) : zipHarsh [3,4] [6,7]]
= sum [x * y | (x, y) <- (2, 5) : (3, 6) : zipHarsh [4] [7]]
= sum [x * y | (x, y) <- (2, 5) : (3, 6) : (4, 7) : zipHarsh [] []]
= sum [x * y | (x, y) <- (2, 5) : (3, 6) : (4, 7) : []]
= sum [x * y | (x, y) <- (2, 5) : (3, 6) : [(4,7)]]
```

```

= sum [x * y | (x, y) <- (2, 5) : [(3,6), (4,7)]]
= sum [x * y | (x, y) <- [(2,5), (3,6), (4,7)]]
= sum [10,18,28]
= 56

```

The current implementation of list comprehensions expects generators to be represented in the AST with an `HsExplicitList`. However in this case because `[(2,5), (3,6), (4,7)]` is a result of GHC evaluation it is represented as the string `"[(2,5), (3,6), (4,7)]"` within an `HsVar` node. This causes a failure when attempting to reduce the list comprehension, so the entire list comprehension is reduced directly to normal form instead. This results in skipping directly to `sum [10,18,28]`, rather than showing the full series of reduction steps. While the list comprehension implementation could be updated to attempt to deal with this, I consider it better to fix the root cause of the issue. Sections 6.1.1 and 6.1.3 outline two future developments which together should fix this problem.

5.3 Type variable issues

The type variable issue discussed in section 4.7 causes a failure for the ‘typeissue’ test, which is as follows:

```

length :: [a] -> Int
length (x:xs) = 1 + (length xs)
length [] = 0

```

```
length [[]]
```

In the formal-actual mapping system the following function is generated:

```

formalactualfunlength :: Show a => [a] -> [(String, String)]
formalactualfunlength (x:xs) = [("x", show x), ("xs", show xs)]

```

Evaluating `formalactualfunlength [[]]` via GHC should return the formal-actual map for the `length [[]]` function application. However in this case this fails, GHC is unable to derive the type of `a`, and therefore ensure it is a member of `Show`. This failure occurs because GHC has no non-empty instances of `a`, for all it knows `a` may not actually be showable! This results in a compiler error, and the GHC evaluation fails. This prevents the map from being generated, which means the reduction fails, this should be resolved by the replacement for the formal-actual mapping system described in section 6.1.1.

5.4 Pattern matching against functions

The issue with pattern matching against functions as seen in section 4.7 causes the test ‘compose’ to fail, which is as follows:

```

compose :: [(a -> a)] -> a -> a
compose [] = id

```

```
compose (f:fs) = f . compose fs
```

```
compose [(map ((+) 1)), (map ((* 2))] [1,2,3]
```

In the formal-actual mapping system the following function is generated:

```
formalactualfunccompose :: Show a -> [(a -> a)] -> [(String, String)]
formalactualfunccompose (f:fs) = [("f", show f), ("fs", show fs)]
```

Evaluating `formalactualfunccompose [(map ((+) 1)), (map ((* 2))] [1,2,3]` via **GHC** should return the formal-actual map for the

`compose [(map ((+) 1)), (map ((* 2))] [1,2,3]` function application. This fails as it requires the evaluation of `show (map ((+) 1))`, which as Haskell functions are not members of `Show` results in a type error. Therefore the map cannot be generated and the reduction fails. This should be resolved by the replacement for the formal-actual mapping system described in section 6.1.1.

5.5 Incorrect operator precedence

As mentioned in section 4.2 the parser does not properly respect mathematics operator precedence. One such example is the ‘operators’ test, in which the initial expression is $2 + 3 * 4$. This is reduced as:

$$\begin{aligned} & 2 + 3 * 4 \\ = & 5 * 4 \\ = & 20 \end{aligned}$$

As Haskell follows standard operator precedence the result should (and if executed with **GHCi** is) be 14! So the reduction should instead be:

$$\begin{aligned} & 2 + 3 * 4 \\ = & 2 + 12 \\ = & 14 \end{aligned}$$

This occurs because the **GHC** parser parses all infix applications as left-associative, resulting in the above situation. So the expression is parsed as $(2 + 3) * 4$ rather than $2 + (3 * 4)$. Details of a possible remedy are detailed in section 6.1.4.

5.6 Support for patterns in comprehensions

As discussed in section 4.4.4 pattern matching is not supported within the binds of a list comprehension. The formals within the pattern will not be replaced with those from the list feeding the bind. This is seen in test ‘listpattern’, in which the comprehension is:

```
[x | (x,y) <- [( 'H' , 0), ( 'e' , 1) ]]
```

This results in the following reduction steps:

$$[x \mid (x, y) \leftarrow [('H' , 0), ('e' , 1)]]$$


```

= [x] ++ [x | (x, y) <- [('e', 1)]]
= [x] ++ [x]

```

Here the reduction of `[x] ++ [x]` fails, because the formal variables have no values bound to them it cannot be evaluated by GHC. This causes the reduction to halt early, although due to the failure to replace the formal variables with actual values it was already nonsensical. This issue will be resolved by applying the new formal-actual mapping system detailed in section 6.1.1.

5.7 Lack of feature support

The remaining issues are caused by a lack of support for various Haskell features. In the case of `lect05-4`, `lect09-1`, `lect10a-3`, `lect14-1`, and `'sumcase'` this results in the reduction failing completely. This is due to a lack of support for redefining operators, custom data types, function composition, `where`, and `case` statements respectively.

Additionally in the case of `lect10a-2` the reduction doesn't fail, it simply skips detail. The code in question is a lambda expression:

```
(\x -> x > 0) 3
```

And the output is:

```

(\ x -> x > 0) 3
= True

```

Whereas the expected output would be something like this:

```

(\ x -> x > 0) 3
= 3 > 0
= True

```

As lambda expressions are not yet supported, the stupid computer interprets it via GHC resulting in this skipped step. The issues outlined in this section will be rectified simply with more development time.

Chapter 6

Conclusions

The main achievement of the project so far has been implementing the core functionality of the stupid computer. The current program has many limitations, but these can be overcome now that core functionality is in place.

6.1 Future developments

This section contains a number of more detailed possible resolutions to some of the limitations seen previously.

6.1.1 Representing call-by-name

Currently the stupid computer cannot represent Haskell's call-by-name reduction strategy, it would be useful for users if this feature could be properly represented. A major reason for this issue is that the current formal-actual mapping system forces the evaluation of all arguments before they are passed into functions. As seen previously this prevents the evaluation of fairly common Haskell programs, for example using `zip` to index a string.

```
zip "Hello, World!" [0..]
```

Presently the formal-actual map will attempt to evaluate `[0..]`, which (being an infinite list) takes an infinite amount of time. Were call-by-name implemented it would be possible for a user to get a trace as follows:

```
zip "Hello, World!" [0..]
= ('H',0) : zip "ello, World!" [1..]
= ('H',0) : ('e',1) : zip "llo, World!" [2..]
= ('H',0) : ('e',1) : ('l',2) : zip "lo, World!" [3..]
...
```

The key to enabling this will be replacing the formal-actual mapping system. This new system will be based on matching the AST of an argument with the AST of the

patterns to achieve the mapping. Figure 6.1 illustrates this for the pattern $(x:xs)$ and the argument $[1, 2, 3, 4]$.

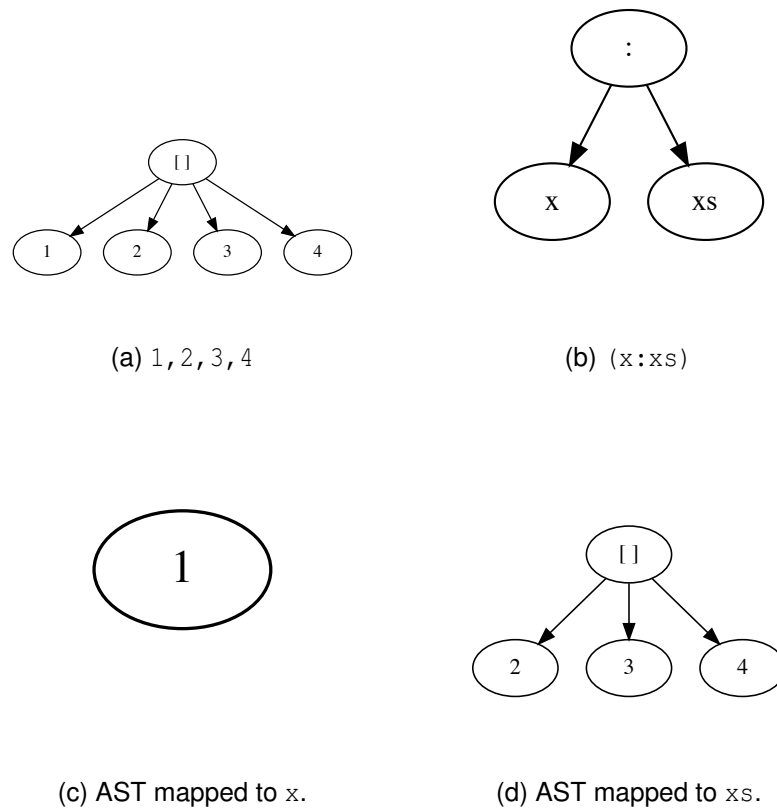


Figure 6.1: (simplified) AST view of planned formal actual mapping process

This will prevent the arguments from needing to be evaluate with GHC, so call-by-name can be shown. No longer relying on GHC interpretation also has other benefits. Firstly the AST structure of arguments will no longer be lost, for example this will allow list comprehensions to use function results as generators. It will also provide a fix to the issue with type variables highlighted in section 5.3, as there will no longer be any type derivation that can fail. Finally I anticipate an improvement in performance thanks to the elimination of costly GHC interpretation.

6.1.2 Representing Sharing

Sharing is another property of Haskell that would be useful to represent. This will be especially useful to implement once a call-by-name strategy has been implemented. To show the issue with purely implementing call-by-name without sharing consider the following two examples, one shows the program as it works now, the other shows how it may work with call-by-name.

```
square (1 + 4)
= square 5
```

```

= 5*5
= 25

square (1 + 4)
= (1 + 4) * (1 + 4)
= 5 * (1 + 4)
= 5*5
= 25

```

The computation of $(1 + 4)$ happens twice! The implementation of sharing would avoid this, it may look a little like this:

```

square (1+4)
= let x = (1+4)
  in x*x
= let x = 5
  in x*x
= 5*5
= 25

```

Here call-by-name has been followed, and $(1 + 2)$ only has to be calculated once. This reduction strategy is also known as call-by-need [11].

6.1.3 Improvements to GHC interpretation

Currently when the stupid computer falls back to GHC interpretation, such as when collapsing a function call to normal form, the result has no AST structure. This can be problematic, especially with the proposed formal-actual mapping system! Therefore we must try and keep this structure whenever possible, the approach to this will be twofold.

Firstly very common operations (i.e $+$, length , $++$ etc) can be directly implemented, avoiding GHC interpretation altogether. Not only will this allow the structure to be kept, it should improve performance. Though it will not be possible to implement this for all operations, some will still have to go to GHC. To try and reclaim this structure I intend to run the results of the interpreter through the GHC lexer and parser. The AST representing the results can then be used instead of a flat string. While this will be quite an expensive operation, it will give the stupid computer far more flexibility.

6.1.4 Fixing operator precedence

As seen in section 5.5 currently infix operator precedences are not dealt with properly, this for example leads to $2 + 3 * 4$ being treated as $(2 + 3) * 4$ not $2 + (3 * 4)$. These are resolved in GHC by the renamer [13]. As seen in section 4.2 the main purpose of the renamer is not fixing operator precedences, but fully resolving identifiers. So for example `sum` becomes `Data.Foldable.sum`, and `+` becomes `GHC.Num.+`. Unfortunately it does not appear to be possible to do the re-association without the rest of the renaming process. I currently have two possible approaches to fix the issue:

- One approach to fixing the precedences would be to apply the standard GHC renamer to the AST, and then undo the identifier resolution. This would be an ideal solution, as it would avoid reimplementing functionality which already exists within GHC. However I do not yet know the feasibility of this solution, and to do so will take further research into the renamer. Unfortunately I only discovered this issue with the operator precedence parsing during my evaluation of the program, so have not as of yet been able to complete this.
- If doing this un-renaming is infeasible then an alternative approach will be to re-implement the method used by the GHC renamer directly within my own implementation, such that no renaming takes place at all.

6.1.5 User interface improvements

I have two priorities for improving the user interface. The first would be to implement the highlighting of the reduced redex at each step of the reduction, in a similar way to DrRacket (see Section 2.2). This would help users identify which redex has been reduced at each step, something that can be challenging for complex expressions.

Another improvement would be to allow users to use normal Haskell files which only contain definitions. Currently these files must also contain the expression they wish to see reduced, which means the files are not directly loadable by GHCi (see Section 4.9). This may be replaced with a system where users load in a file containing only function definitions, then provide expressions they want to see reduced via the CLI. This would allow students to examine several main expressions without having to modify their input file.

6.1.6 User feedback

Currently I have not acquired any user feedback on the stupid computer, this of course will need to be rectified. I plan to make the tool available to students taking the Informatics-1A course in the 21/22 academic year. It will then be possible to gather feedback from the students, whether through questionnaires, interviews or both.

Bibliography

- [1] The Hint Authors. `hint`. <https://hackage.haskell.org/package/hint>. Accessed on 05.10.2020.
- [2] Ghc pretty printer. <https://hackage.haskell.org/package/ghc-8.10.1/docs/Outputable.html>. Accessed on 01.10.2020.
- [3] Hackage. <https://hackage.haskell.org>. Accessed on 25.01.2021.
- [4] Stack. <https://docs.haskellstack.org/en/stable/README/>. Accessed on 17.02.2021.
- [5] Sum definition. <https://hackage.haskell.org/package/base-4.14.1.0/docs/src/Data.Foldable.html#sum>. Accessed on 09.03.2021.
- [6] Informatics 1a - university of edinburgh. <http://www.drps.ed.ac.uk/20-21/dpt/cxinfr08025.htm>, 2020. Accessed on 21.09.2020.
- [7] Chris Dornan and Isaac Jones. Alex user guide, 2003.
- [8] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
- [9] Robert Bruce Findler. Drracket: The racket programming environment. *Racket Language Documentation*, 2014.
- [10] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.
- [11] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.
- [12] Simon Marlow and Andy Gill. Happy user guide. *Glasgow University, December, 1997*.
- [13] Simon Marlow, Simon Peyton Jones, et al. The glasgow haskell compiler, 2004.
- [14] Donald Sannella, Michael Fourman, Haoran Peng, and Philip Wadler. *Introduction to Computation*. 2020.
- [15] The GHC Team. The ghc api. <https://hackage.haskell.org/package/ghc>. Accessed on 05.10.2020.

- [16] The GHC Team. Hsexpr ast documentation. <https://hackage.haskell.org/package/ghc-8.10.1/docs/GHC-Hs-Expr.html>. Accessed on 09.04.2021.
- [17] The GHC Team. multi-way-if. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/multiway_if.html. Accessed on 04.11.2020.
- [18] The GHC Team. Ghci user's guide. https://downloads.haskell.org/ghc/latest/docs/html/users_guide/. Accessed on 10.04.2021.
- [19] The GHC Team. Prelude. <https://hackage.haskell.org/package/base-4.14.1.0/docs/Prelude.html>. Accessed on 09.03.2021.
- [20] D. A. Turner. Recursion equations as a programming language. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, 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, 2016.
- [21] Chris Uehlinger. Lambda bubble pop. <https://chrisuehlinger.com/LambdaBubblePop>. Accessed on 07.09.2020.