University of Canterbury

Department of Computer Science

COSC461 2017 Final Report

Lazy Execution in Imperative Programming Languages

Logan Glasson supervised by Dr. Walter Guttmann

Abstract

We introduce and specify an imperative programming language featuring "lazy execution", an imperative analogue to functional programming's lazy evaluation. We investigate common patterns and higher level constructs to simplify programs in such a language. Some behaviours of programs are then measured to demonstrate that they are indeed being lazily executed.

Contents

1 Introduction **3**

2 Background **5**

2.1 Lazy Evaluation 5

2.1.1 Efficiency 5

2.1.2 Infinite Data Structures 6

2.1.3 Abstraction 6

2.2 Functional and Imperative Languages 7

2.3 Comparable Features 8

2.3.1 Short-circuiting operators 8

2.3.2 Conditionals 8

2.3.3 Iterators and Generators 9

2.3.4 Closures 9

2.4 Operational Semantics 9

3 Approach **11**

4 Description of the Language **13**

4.1 Core Language 13

4.1.1 Notation 13

4.1.2 Initial Grammar 14

4.1.3 `Load` 14

4.1.4 `Assign` 15

4.1.5 `UnOp` 16

4.1.6 `BinOp` 17

4.1.7 Example 18

4.1.8 `If` 19

4.1.9 `Call` . . . 20
4.1.10 Example . . . 21
4.1.11 `Expose` . . . 23
4.1.12 `Tag` . . . 24
4.2 Higher Level Constructs . . . 26
4.2.1 Expressions . . . 27
4.2.2 Linked Lists . . . 27
4.2.3 Arbitrary Data Structures . . . 28
4.2.4 Function Definition . . . 28
4.2.5 Finite Loops . . . 29
4.2.6 Infinite Data Structures . . . 29
4.3 Errors . . . 31

5 Results **32**
5.0.1 Sorting . . . 32

6 Discussion **34**
6.1 Is this Language Imperative? . . . 34
6.2 Applications . . . 35
6.3 Implementation . . . 35
6.4 Usability . . . 36

7 Conclusion **37**

Bibliography **38**

A Full Language Description **40**
A.1 Grammar . . . 40
A.2 Reduction Rules . . . 41

1 Introduction

Lazy evaluation is a feature of some programming languages, under which computations are deferred until their results are needed. Instead of performing a computation up-front when a statement is encountered, some sort of structure is stored which describes the computation that should be performed when the resulting value is needed[1]

Lazy evaluation introduces several interesting possibilities. First, there are cases where lazy evaluation can improve performance without a programmer needing to manually optimise code. Second, lazy evaluation allows for infinite data structures to be described and manipulated. An example of this would be describing a list of primes which contains every prime number in order. The contents of such a data structure are only computed as required by other parts of a program. Third, there are some extra opportunities for abstraction in lazy programming languages.

Existing lazy programming languages all fall within the functional paradigm. The goal of this research was to investigate how laziness could be applied in an imperative programming language. There are a few factors which motivated this topic. First, there are several potential benefits to lazy programming languages. Second, imperative languages are more widespread and more intuitive to many programmers. Third, there is a general trend of features from functional languages entering imperative languages. Finally, this topic is, to date, relatively unexplored, and investigating it satisfies academic curiosity.

We have specified a simple imperative programming language and an operational semantics for its lazy evaluation. We have also investigated higher level constructs which might improve subjective usability of a lazy imperative language. Finally, we give some results on the performance of this lazy imperative language.

[1] This is a simplification, and there is a significant amount of research into compiling lazy programs.

The scope of this research is primarily theoretical, and it was not carried out with any specific applications in mind. Optimising the demonstration language implementation for time or memory performance was not a goal of this research.

In this report, we first provide some background information on lazy evaluation and relevant programming languages ideas. We then briefly outline the approach we took, before describing in detail the lazy imperative language we produced. We then give some results on the behaviours of simple programs and the performance of algorithms. Finally, we give some discussion about the research. The appendix features a concise definition of the language and a code listing of the current demonstration interpreter.

2 Background

2.1 Lazy Evaluation

Lazy evaluation is "an execution mechanism in which an object is evaluated only at the time when, and to the extent that, it is needed"[1]. It is sometimes referred to as "call-by-need", meaning that parameters to functions are only computed as they are needed. The effect of this evaluation strategy is that computations are deferred until their results are actually needed.

Lazy evaluation gives several interesting effects which we examine in more detail: potential increases in efficiency, the ability to define and manipulate infinite data structures, and potential opportunities for abstraction.

2.1.1 Efficiency

There are some programs which are more efficient when run in a lazy way. An example of this is the behaviour of Haskell's `sort` function. The expression `head . sort` describes sorting a list and then taking its "head" (its first element) to find the minimum element of the original list. Evaluated in a non-lazy manner, this would be expected to take $\mathcal{O}(n \log n)$ where n is the length of the list, due to the complexity of sorting [2]. Evaluated in a lazy manner, this instead takes just $\mathcal{O}(n)$, because only the first element is requested. This is the best complexity for finding the minimum of a list, because every element has to be examined at least once. This behaviour does depend on the implementation of the `sort` function, but it demonstrates the potential for lazy evaluation to improve the performance of programs. This behaviour is explained in more detail at [3]

This sort of improvement in efficiency could also occur in cases where many expensive interdependent computations are performed, but only some (unknown at compile time) set of results are actually needed. Rather than manually implement

an algorithm to determine which computations are required, a programmer could just naively implement the computation and let the laziness of the programming language cull all but the required operations.

2.1.2 Infinite Data Structures

One major effect that lazy evaluation gives is the ability to describe and manipulate "infinite data structures". These are data structures which behave as though they are infinitely large, by only computing their contents as required.

An example of this is seen in Haskell, where a simple data type is that of a "linked list". A linked list object is either a pair of a head value and a tail list, represented by the cons operator "`:`", or the empty list, represented by "`[]`". For instance, the sorted list of positive integers less than four could be represented as `(1:(2:(3:[])))`. It is possible to then write a function which returns a list where the tail of the list is the result of a call to the function itself, such as `f = (1:f)`. In this function `f`, a list is returned with the integer 1 as the head and the result of calling `f` as the tail. Evaluating this program in a non-lazy language would not terminate due to the endlessly recursive call to `f`, but in a lazy language the function call is only performed when the tail of a list is needed and so this infinite list can be manipulated similarly to any other list [4, 5].

The previous example was very simple, but this idea of describing an infinite data structure has many more complex uses. For example, it is possible to describe a list of increasing integers, or a list which indefinitely repeats another list, or a list containing all prime numbers. There are also uses beyond just lists. The same idea can be applied to produce infinite trees, or graphs.

2.1.3 Abstraction

Lazy execution in imperative languages has the potential to benefit programmers. Hughes [6] argues that "modularity is the key to successful programming", and gives an example to demonstrate the modularity afforded by lazily evaluated programs. The state tree of a game (such as chess) can be represented by a lazily defined, potentially infinite, tree. This can then be manipulated by a series of functions, such as to take heuristic scores of each state, limit the tree to a finite depth, and perform the alpha-beta algorithm to find a good move. All of these functions could be written independently of each other. The point is made that

in an imperative language, the depth limiting code would be necessarily coupled to the tree generation code. Additionally, to perform further manipulations either the entire tree would need to be kept in memory, or these further manipulations would have to be coupled to the monolithic tree generation code.

Abstraction is often regarded as a desirable feature of a codebase, as a way to manage complexity and reduce the costs of software development and maintenance [7]. This provides motivation to explore methods of abstraction, such as lazy evaluation.

2.2 Functional and Imperative Languages

There is no well established definition of the difference between functional and imperative programming languages, and the paradigms which a language belongs to are largely determined by programming language creators and users on subjective criteria. There are, however, some generally agreed upon features and properties of languages that suggest that they may be either functional or imperative.

Van Roy [8] gives a taxonomy of 27 programming paradigms by a set of 18 features. Under this taxonomy the functional and imperative paradigms are distinguished by the presence of closures for functional languages, and the presence of state for imperative languages. Jordan et al [9] compares 5 programming paradigms with features from 9 categories. The distinction between the functional and imperative paradigms is largely in categories to do with state and input/output.

In this paper, we distinguish between functional and imperative languages primarily by the presence of "state" and "statements". Languages such as Haskell or Lisp are comprised of nested expressions rather than sequences of statements, and state is largely contained within those expressions, which suggests that they are functional. Languages such as C++, Python, or Java are comprised of sequences of statements, and have state which is maintained separately from their code, which suggests that they are imperative.

One notable observation is the trend of some features from functional languages entering modern imperative languages. This is seen in recent releases of C++, Java, Python, and in newer languages such as Rust and Swift. C++11 introduces range based for loops[10], and lambda functions and expressions [11].

Java 8 in 2014 also introduced lambda expressions. Python 3 features lambdas, iterators and generators, and list comprehension, among other things [12]. Rust, a language developed since 2010 and sponsored by Mozilla Research, is a systems programming language with strong restrictions on mutability (the ability to change values in place), type classes inspired by Haskell, iterators and generators, and lambda functions [13]. Swift, developed by Apple Inc., also restricts mutability, and features function chaining, composition, and partial application — staples of functional programming languages.

2.3 Comparable Features in Existing Imperative Languages

There are some existing features common to imperative languages which give behaviours similar to those seen in lazy languages. These features and their limitations are briefly discussed here.

2.3.1 Short-circuiting operators

Short-circuiting operators are found in expressions in many imperative languages, with common examples being boolean operators or the ternary operator, such as `&&`, `||`, and `?` in C++. They are short-circuiting in that in some cases the first argument fully determines the result, and the remaining arguments are not computed. For instance, the C++ expression `false && infinite_loop()`, where `infinite_loop` is a function which loops indefinitely, would evaluate to false without entering an infinite loop.

This behaviour is lazy because parts of computation are not carried out unless they are needed. However, this is a very specific case, and it only occurs within expression.

2.3.2 Conditionals

Conditionals such as `if` ... `then` ... `else` ... statements can be considered a simple form of lazy evaluation. This is because depending on the result of some condition, only one block of code will be executed. The results produced by the

other block are not needed, and are not computed. Again, this is only a very specific case of laziness, but is seen in the majority of programming languages.

2.3.3 Iterators and Generators

Some imperative languages, such as Python, Rust, have the concept of "iterators" and "generators", which can be used to produce results only when they are needed. The exact behaviour of these structures varies between the languages which implement them. In general, a generator allows some function to be specified which produces values on demand. A generator might give an increasing list of numbers, or give a list of permutations of some string in lexicographical order. A generator typically implements the interface of an iterator, which might allow functionality such as mapping (applying a function to each element), filtering (removing elements depending on a predicate on their values), or looping through with a `for ... in` statement.

2.3.4 Closures

Closures are dynamically generated functions, which inherit the scope at the point of their definition. They were first developed in association with the λ-calculus [14]. Since then they have appeared in many imperative programming languages. Closures can be used to implement an explicit form of laziness. This is done by wrapping every computation in a closure. This closure should perform the computation the first time it is called, and subsequently return a cached value of the result.

2.4 Operational Semantics

An operational semantics is a formal description of the computational process of running a program in a particular language. One can be used to prove results about the behaviours of programs, such as correctness, safety, or performance. In this report we give a structural operational semantics, which is a particular type of operational semantics that is structural — it focuses on the syntactic structure of the language. Structural operational semantics are introduced by Plotkin [15], and good introductions can be found in [16] and [17].

At its core, an operational semantics is a set of rules which describe the computations that should be performed to execute any program. The rules we give here are of the following form, where $\langle precondition \rangle$ and $\langle postcondition \rangle$ are predicates on the program and its state, and the notation {...} expresses that there may be zero or more preconditions.

$$\frac{\{\langle precondition \rangle\}}{\langle postcondition \rangle}$$

A rule of this form can be interpreted as "if the preconditions can be met, then the postcondition holds". The specific notation for a predicate is generally given as part of the operational semantics, as different languages may require different predicates.

3 Approach

> I choose a lazy person to do a hard job. Because a lazy person will find an easy way to do it.
>
> Bill Gates

The work undertaken for this project was rather open-ended and there was no predetermined path to follow. The goals of this research were to produce a working demonstration of a lazily executed imperative programming language, investigate properties of this language, develop a formal definition for an imperative lazy execution strategy, and give results about the behaviour of imperative lazy programs.

The nature of the work was largely investigative and innovative because there was little prior work on laziness in imperative languages. This led to an iterative and experimental approach, where the demonstration interpreter evolved over the course of the project. In the first stages of research, this evolution was motivated by attempting to implement various examples and algorithms in the language and identifying patterns and problems. In the later stages, the evolution was motivated more by goals such as providing intuitive high level concepts and defining a formal operational semantics.

There are several ways in which this research is evaluated:

Ability to Implement Examples: We can attempt to implement examples of various algorithms and subjectively evaluate the feasibility of the language.

Performance of Algorithms: We can measure the performance of various algorithms and compare with well-studied traditional imperative implementations. Note that real-time performance is not a goal of this research, and so

comparisons are instead made in terms of time-independent metrics such as the number of operations.

In this report, we primarily present the final product of this research, although as we introduce the language in section (4) we discuss some motivations behind decisions and alternatives that were considered.

4 Description of the Language

4.1 Core Language

In this section we gradually introduce the core details of the language, incrementally building towards a complete definition. At a high level, a *program* in the language is a sequence of *statements*, which manipulate *values* held by various *variables* and control the flow of the program with conditionals and recursion.

Traditionally, an imperative language performs a computation when a statement is first encountered. The key difference between that behaviour and the lazy behaviour given here is that the computation is instead motivated by the desired output variable, which we refer to as the *goal*. To find the value of this goal, the program is processed in a reverse order — the last statement in a program definition is the first statement considered during execution.

In the following subsections, we express the grammar and behaviour, give high level interpretations, and explain the motivation for inclusion of the various details and statements. We express the grammar of programs using the Extended Backus-Naur Form (EBNF) notation. The grammar that we give describes both how programs could be written by a programmer, and the internal representation of a program. We introduce an additional notation of $\|$, to represent an alternative which is only accessible internally, and not by the programmer. We express the behaviour using a formal operational semantics, as introduced in section 2.4. This can be used both to mathematically derive results about the language, and to give an implementation of an interpreter for the language.

4.1.1 Notation

We use an original notation in our operational semantics to describe the execution of a program in the language. A basic unit of execution takes a program and

a goal, and returns an updated program and a value. The updated program is part of the result because in the process of execution the program may be manipulated, such as by expanding conditionals or calls. By updating the program, unnecessary recomputation is avoided. This basic unit of execution can be represented by a function, mapping from a tuple of a program and a goal to a tuple of a program and value. We express that function using the following notation.

$$\Big(\langle program\rangle \Uparrow \langle goal\rangle\Big) \rightarrow \Big(\langle program\rangle \Downarrow \langle value\rangle\Big)$$

When executing the program on the left hand side, for the goal, the result is program on the right hand side and the value.

4.1.2 Initial Grammar

First we introduce the basic elements of the grammar. A program is a sequence of statements. And identifier is an alphanumeric string beginning with a lowercase letter. A goal is, for the time being, a single identifier. A value is, for the time being, either an integer or a boolean. These definitions will be extended later, as concepts are introduced to the language.

$$\begin{aligned}
\langle program\rangle &::= \{\langle statement\rangle\} \\
\langle identifier\rangle &::= \langle lowercase\rangle\{\langle alphanumeric\rangle\} \\
\langle goal\rangle &::= \langle identifier\rangle \\
\langle value\rangle &::= \langle integer\rangle \\
&\quad\mid \langle boolean\rangle
\end{aligned}$$

4.1.3 `Load`

The first statement that we introduce is **`Load`**, which is used to load constant values into variables for later use. If we execute a program where the most recent statement affecting a variable is a **`Load`** statement, we would expect to find the value held by that variable to be the value specified by that **`Load`** statement. The **`Load`** statement should not affect other variables at all. We extend our grammar and operational semantics as follows,

$$\langle \textit{statement} \rangle ::= \langle \textit{statement} \rangle$$
$$\mid \texttt{Load}\ \langle \textit{identifier} \rangle\ \texttt{<-}\ \langle \textit{value} \rangle\texttt{;}$$

$$\frac{x = g}{\Big(P\texttt{;Load}\ x\ \texttt{<-}\ v\texttt{;} \Uparrow g\Big) \rightarrow \Big(P\texttt{;Load}\ x\ \texttt{<-}\ v\texttt{;} \Downarrow v\Big)} \quad \text{(load.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\Big) \rightarrow \Big(P' \Downarrow r\Big)}{\Big(P\texttt{;Load}\ x\ \texttt{<-}\ v\texttt{;} \Uparrow g\Big) \rightarrow \Big(P'\texttt{;Load}\ x\ \texttt{<-}\ v\texttt{;} \Downarrow r\Big)} \quad \text{(load.2)}$$

The rule (load.1) describes what to do with a program that looks like $P\texttt{;Load}\ x\ \texttt{<-}\ v\texttt{;}$, when finding the value of g, in the case where $x = g$. This means that the goal g is the same as the destination variable x of the **Load** statement, and so we give the result v, the value specified in the **Load** statement.

The rule (load.2) describes the case where $x \neq g$. In this case, the value of g should be found in the preceding program, as the **Load** statement has no effect. This execution in the preceding program is shown in the second precondition, which finds the value of g under the preceding program P, putting the resulting program into P' and the resulting value into r. The value r is returned directly, and the program is updated with P' so that any modifications to the preceding program are preserved.

4.1.4 Assign

The next statement that we introduce is **Assign**, which is used to load the value of one existing variable into another variable. If we execute a program which tries to find the value of a variable assigned to by an **Assign** statement, we should find the value of the right hand side of the **Assign** statement in the preceding program, and give that as the result. To include **Assign**, we extend our grammar and operational semantics as follows,

$$\langle statement \rangle ::= \langle statement \rangle \\ \mid \texttt{Assign}\ \langle identifier \rangle\ \texttt{<-}\ \langle goal \rangle\texttt{;}$$

$$\frac{x = g \qquad \Big(P \Uparrow y\Big) \rightarrow \Big(P' \Downarrow r\Big)}{\Big(P\texttt{;Assign}\ x\ \texttt{<-}\ y\texttt{;} \Uparrow g\Big) \rightarrow \Big(P'\texttt{;Load}\ x\ \texttt{<-}\ r\texttt{;} \Downarrow r\Big)} \qquad \text{(assign.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\Big) \rightarrow \Big(P' \Downarrow r\Big)}{\Big(P\texttt{;Assign}\ x\ \texttt{<-}\ y\texttt{;} \Uparrow g\Big) \rightarrow \Big(P'\texttt{;Assign}\ x\ \texttt{<-}\ y\texttt{;} \Downarrow r\Big)} \qquad \text{(assign.2)}$$

The rule (assign.1) covers the case where $x = g$, and so the goal matches the destination of this assignment. The rule searches the preceding program for the goal specified in the right hand side of the assignment, and gives this as the result. The resulting program is also updated to have a **`Load`** statement rather than an assignment. The means that any future execution searching for the same goal will terminate at this statement and not continue to search in the preceding program a second time.

The rule (assign.2) covers the case where $x \neq g$, and so the goal does not match the variable assigned to. In this case, the value of g should be found in the preceding program, as the **`Assign`** statement has no effect. This is similar to the rule (load.2), in that the value is found in the preceding program and the returned program is updated.

4.1.5 `UnOp`

The next statement that we introduce is **`UnOp`**, which is used to perform a unary operation on one existing variable's value and put the result into another variable. If we execute a program which tries to find the value of a variable assigned to by a **`UnOp`** statement, we should find the value of the operand, perform the operation, and give that as the result. To include **`UnOp`**, we extend our grammar

and operational semantics as follows,

$$\langle \textit{unary-operator} \rangle ::= \texttt{!} \mid \texttt{-}$$

$$\langle \textit{statement} \rangle ::= \langle \textit{statement} \rangle \mid \texttt{UnOp}\ \langle \textit{identifier} \rangle\ \texttt{<-}\ \langle \textit{unary-operator} \rangle\ \langle \textit{goal} \rangle\texttt{;}$$

$$\frac{x = g \qquad \Big(P \Uparrow y\Big) \to \Big(P' \Downarrow r\Big) \qquad r' = opr}{\Big(P\texttt{;UnOp}\ x\ \texttt{<-}\ op\ y\texttt{;} \Uparrow g\Big) \to \Big(P\texttt{;Load}\ x\ \texttt{<-}\ r'\texttt{;} \Downarrow r'\Big)} \tag{unop.1}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{;UnOp}\ x\ \texttt{<-}\ op\ y\texttt{;} \Uparrow g\Big) \to \Big(P'\texttt{;UnOp}\ x\ \texttt{<-}\ op\ y\texttt{;} \Downarrow r\Big)} \tag{unop.2}$$

The **UnOp** statement is similar to the **Assign** statement in that it finds a value in the preceding program. It is different in that in the case of $x = g$ it performs a unary operation on the result before returning it. The unary operation is either binary negation or integer negation. In the case of type mismatching, such as attempting to find the binary negation of an integer, the rule does not apply and so the program does not execute, as discussed later in section (4.3).

4.1.6 BinOp

The last statement before we give an example is **BinOp**, which is used to perform a binary operation on two existing variables' values and put the result into another variable. If we execute a program which tries to find the value of a variable assigned to by a **BinOp** statement, we should find the values of both operands, perform the operation, and give that as the result. To include **BinOp**, we extend our grammar and operational semantics as follows,

$$\langle \textit{binary-operator} \rangle ::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{==}$$

$$\langle \textit{statement} \rangle ::= \langle \textit{statement} \rangle \mid \texttt{BinOp}\ \langle \textit{identifier} \rangle\ \texttt{<-}\ \langle \textit{goal} \rangle\ \langle \textit{binary-operator} \rangle\ \langle \textit{goal} \rangle\ \texttt{;}$$

$$\frac{x = g \quad \Big(P \Uparrow y\Big) \to \Big(P' \Downarrow r_1\Big) \quad \Big(P'' \Uparrow z\Big) \to \Big(P' \Downarrow r_2\Big) \quad r' = r_1\ op\ r_2}{\Big(P\texttt{;BinOp}\ x\ \texttt{<-}\ y\ op\ z\texttt{;} \Uparrow g\Big) \to \Big(P''\texttt{;Load}\ x\ \texttt{<-}\ r'\texttt{;} \Downarrow r'\Big)} \quad \text{(binop.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{;BinOp}\ x\ \texttt{<-}\ y\ op\ z\texttt{;} \Uparrow g\Big) \to \Big(P'\texttt{;BinOp}\ x\ \texttt{<-}\ y\ op\ z\texttt{;} \Downarrow r\Big)} \quad \text{(binop.2)}$$

The **BinOp** statement is similar to the **UnOp** statement in that it finds values, performs and operation on them, and returns the result. It is different in that in the case of $x = g$ it loads two values instead of just one. To load two values, it uses an intermediate program result P' as well as a final program result P'', again to avoid duplicate calculations. In the case of type mismatching or arithmetic error, such as division by zero, the rule does not apply and so the program does not execute, as discussed later in section (4.3).

4.1.7 Example

Now that we have defined **Load**, **Assign**, **UnOp**, and **BinOp**, we can give a simple example program and discuss its behaviour.

```
1 Load x <- 42;
2 Load one <- 1;
3 Load two <- 2;
4 BinOp three <- two + one;
5 BinOp six <- three + three;
6 Assign ono <- six;
```

Interpreted in a non-lazy manner, the above program loads constant values into the variables `x`, `one`, and `two`, then adds them to produce the variable `three`, then adds that to itself to produce the variable `six`, and finally assigns that value to the variable `ono`.

Interpreted in the lazy manner described by the rules above, we need to specify a goal to find the value of. If we were to specify `ono` as our goal, the execution would first consider the final **`Assign`** statement. As the left hand side of the assignment matches the current goal, the rule (assign.1) is applied. Now, the value of `six` is searched for in the program consisting of lines [1-5]. The execution now considers the **`BinOp`** statement on line 5, and the rule (binop.1) is applied. First, this triggers a search for the value of `three` in the program consisting of lines [1-4], which eventually returns the value 3 along with an updated program. As `three` is found by the rule (binop.1), the updated program has the **`BinOp`** replaced by a **`Load`**, and so looks like the following:

```
1 Load x <- 42;
2 Load one <- 1;
3 Load two <- 2;
4 Load three <- 3;
```

At this point, the **`BinOp`** statement on line 5 begins another search for the value of `three`, as the second operand, in the updated program. Because the updated program has cached the value of `three`, this search returns straight away, without needing to look back to lines 2 and 3. The **`BinOp`** finally returns the value 6 for the variable `six`, and the **`Assign`** returns the same value 6 for the variable `ono`.

This example demonstrates that values are not calculated until they are needed, and that once a value has been found execution does not need to continue. Note that the statement on line 1 is never reached because all required variables are defined after it.

4.1.8 `If`

We now introduce the first control flow statement, **`If`**, which is used to conditionally execute a subprogram (a sequence of statements), dependant on the value of a variable (the condition). If a program's execution encounters a conditional, it should first determine whether the condition is true or false, and then depending

on the result, possibly execute the subprogram in the place of the `If` statement. To include `If`, we extend our grammar and operational semantics as follows,

$$\begin{aligned}\langle \textit{statement} \rangle ::= \; & \langle \textit{statement} \rangle \\ & \mid \texttt{If} \;\; \langle \textit{goal} \rangle \;\; \texttt{then} \;\; \langle \textit{program} \rangle \texttt{;}\end{aligned}$$

$$\frac{\Big(P \Uparrow y\Big) \rightarrow \Big(P' \Downarrow \texttt{true}\Big) \quad \Big(P'\texttt{;}Q \Uparrow g\Big) \rightarrow \Big(P'' \Downarrow r\Big)}{\Big(P\texttt{;If } \; y \; \texttt{ then } \; Q\texttt{;} \Uparrow g\Big) \rightarrow \Big(P'' \Downarrow r\Big)} \qquad \text{(conditional.1)}$$

$$\frac{\Big(P \Uparrow y\Big) \rightarrow \Big(P' \Downarrow \texttt{false}\Big) \quad \Big(P' \Uparrow g\Big) \rightarrow \Big(P'' \Downarrow r\Big)}{\Big(P\texttt{;If } \; y \; \texttt{ then } \; Q\texttt{;} \Uparrow g\Big) \rightarrow \Big(P'' \Downarrow r\Big)} \qquad \text{(conditional.2)}$$

In both rules, the first precondition finds the value of the condition ys in the preceding program P. The rule (conditional.1) covers the case where the value is found to be `true`. In this case, we want to insert the program described by Q in the place of the `If` statement, and evaluate it for the original goal. This is achieved by the second precondition, which produces a executes the concatenation of P' and Q to find the resultant program P'' and value r. Note that the intermediate program P' is used between finding the value of the condition y and the value of the result r, to avoid duplicate computations.

The rule (conditional.2) covers the case where the value found is `false`. In this case, we disregard the entire `If` statement, and just find the value of the goal g in the preceding program P.

4.1.9 `Call`

The second control flow statement, `Call`, is used to insert the contents of a previously defined subprogram. This can be used to create loops and recursion. To introduce this, we extend our grammar to allow programs as values. The argument to `Call` is a goal, which when found should result in a value representing a program.

This program is inserted in the place of the `Call` statement. To include `Call`, we extend our grammar and operational semantics as follows,

$$\begin{aligned} \langle value \rangle &::= \langle value \rangle \\ &\mid \langle program \rangle \\ \langle statement \rangle &::= \langle statement \rangle \\ &\mid \texttt{Call } \langle goal \rangle \texttt{;} \end{aligned}$$

$$\frac{\Big(P \Uparrow f\Big) \to \Big(P' \Downarrow R\Big) \quad \Big(P'\texttt{;}R \Uparrow g\Big) \to \Big(P'' \Downarrow r\Big)}{\Big(P\texttt{;Call } f\texttt{;} \Uparrow g\Big) \to \Big(P'' \Downarrow r\Big)} \qquad \text{(call)}$$

This rule first finds the value of f to be R, which should be a program. It then executes the concatenation of the programs P' and R to find the value of g. Despite its simple definition, `Call` is the only statement in the language which introduces any sort of looping or recursive behaviour. Unlike the concept of function calls in many languages, this definition of calling leads to "dynamic scoping". This means that the scope of the subprogram that is called is inherited from the call site, rather than the definition site. This leads to some potentially unfamiliar behaviour. In section 4.2 we discuss some higher level constructs to provide more approachable behaviours.

4.1.10 Example

Now that we have introduced the `If` and `Call` expressions, we can give another example:

```
1 Load one <- 1;
2 Load zero <- 0;
3 Load x_to_the_n <-
4   BinOp b <- n == zero;
5   If b then
6     Load r <- 1;;
7   UnOp b <- ! b;
8   If b then
```

```
 9      BinOp n <- n - one;
10      Call x_to_the_n;
11      BinOp r <- r * x;;;
12  Load x <- 2;
13  Load n <- 5;
14  Call x_to_the_n;
```

Note that a double semicolon (`;;`) acts to end a subprogram, and the triple semicolon seen here on line 10 acts to end both the **`If`** and the **`Load`** statements.

The program given here defines a subprogram `x_to_the_n`, which expects variables named `x` and `n` to already exist, and produces the `n`th power of `x` in the variable `r`. It does this by repeatedly calling itself while decreasing the value of `n`, until `n` is equal to zero. When `n` is equal to zero, it sets `r` to one, and then multiplies `r` by `x` once for every call performed. Calling can be though of as expanding the program, and so lines 12-14 would expand to a program like the following:

```
 1  Load n <- 5;                 # n = 5
 2  BinOp b <- n == zero;        # b = false
 3  If b then                    # does not execute
 4    Load r <- 1;;
 5  UnOp b <- ! b;               # b = true
 6  If b then                    # does execute
 7    BinOp n <- n - one;        # n = 4
 8    BinOp b <- n == zero;      # b = false
 9    If b then                  # does not execute
10      Load r <- 1;;
11    UnOp b <- ! b;             # b = true
12    If b then                  # does execute
13      BinOp n <- n - one;      # n = 3
14      ...
15            BinOp n <- n - one;      # n = 0
16            BinOp b <- n == zero;    # b = true
17            If b then                # does execute
18              Load r <- 1;;          # r = 1
19            UnOp b <- ! b;           # b = false
20            If b then                # does not execute
```

```
21              BinOp n <- n - one;
22              Call x_to_the_n;
23              BinOp r <- r * x;;;
24            BinOp r <- r * x;;;      # r = 2
25      ...
26      BinOp r <- r * x;;;    # r = 16
27    BinOp r <- r * x;;;      # r = 32
```

Where the ellipses are used to denote skipping several expansions. It can be seen that the resulting value in `r` is 32, which is the result of 2^5.

4.1.11 Expose

We now introduce the first of two statements dealing with scope. **`Expose`** is used to insert a subprogram which does not affect global variable definitions. This is similar to the concept of local variables in many existing languages. The **`Expose`** statement takes an identifier to expose, and a subprogram to execute. If the **`Expose`** statement is executed with a goal matching the variable it exposes, the subprogram is concatenated to the preceding program and the result is executed to find the value. Otherwise, the **`Expose`** statement skips the subprogram and finds the value in the preceding program. To include **`Expose`**, we extend our grammar and operational semantics as follows, where the notation $\|$ represents an alternative that is for internal use only, and not accessible to the programmer.

$$\begin{aligned}\langle \textit{statement} \rangle ::= \langle \textit{statement} \rangle \\ \mid \texttt{Expose}\ \langle \textit{identifier} \rangle\ \texttt{in}\ \langle \textit{program} \rangle\texttt{;} \\ \| \ \texttt{Marker;}\end{aligned}$$

$$\frac{x = g \quad \textbf{unique marker } m \quad \big(P\texttt{;}m\texttt{;}Q \Uparrow g\big) \to \big(P'\texttt{;}m\texttt{;}Q' \Downarrow r\big)}{\big(P\texttt{;Expose } x \texttt{ in } Q\texttt{;} \Uparrow g\big) \to \big(P'\texttt{;Expose } x \texttt{ in } Q'\texttt{;} \Downarrow r\big)} \quad \text{(expose.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\Big) \rightarrow \Big(P' \Downarrow r\Big)}{\Big(P\texttt{;Expose } x \texttt{ in } Q\texttt{;} \Uparrow g\Big) \rightarrow \Big(P'\texttt{;Expose } x \texttt{ in } Q\texttt{;} \Downarrow r\Big)} \qquad \text{(expose.2)}$$

The rule (expose.1) handles the case where $x = g$, and so we should find the value in the program consisting of the subprogram concatenated to the preceding program. In order to update both the program P and the subprogram Q with the result of the execution step, we need to be able to separate them after we execute their concatenation. We achieve this by generating a unique marker and inserting it between them, and then splitting the resulting program by finding the marker again. Implementation of this is discussed in section (6.3).

The rule (expose.2) handles the case where $x \neq g$ and so we do not consider the subprogram Q at all. The small example below demonstrates the **Expose** statement.

```
1 Load x <- 42;
2 Expose y in
3   Load x <- 21;
4   Assign y <- x;;
```

Finding the value of `y` results in the value 21, while finding the value of `x` results in the value 42. The assignment to `y` in the expose statement is exposed to the outer program, so we see its updated value, while the load to `x` is not exposed and so we only see its original definition.

4.1.12 Tag

The final statement, **Tag**, also deals with scope, and is probably the most complicated statement. It allows a later statement to access variables which were in scope at the **Tag** statement but were then overwritten. To achieve this we extend our definition of a goal in be a sequence of identifiers. To execute with a goal in the form of a sequence, the identifiers should be found in sequence. When a **Tag** for an identifier is found, that identifier should be removed from the goal and execution should continue in the preceding program. To include **Tag**, we extend our grammar and operational semantics as follows,

$$
\begin{aligned}
\langle goal \rangle &::= \langle goal \rangle \\
&\mid \langle identifier \rangle \texttt{.} \langle goal \rangle \\
\langle statement \rangle &::= \langle statement \rangle \\
&\mid \texttt{Tag}\ \langle identifier \rangle \texttt{;}
\end{aligned}
$$

$$
\frac{x = g \qquad \big(P \Uparrow gs\big) \to \big(P' \Downarrow r\big)}{\big(P\texttt{;Tag}\ x\texttt{;} \Uparrow g\texttt{.}gs\big) \to \big(P'\texttt{;Tag}\ x\texttt{;} \Downarrow r\big)} \qquad \text{(tag.1)}
$$

$$
\frac{x \neq g \qquad \big(P \Uparrow g\texttt{.}gs\big) \to \big(P' \Downarrow r\big)}{\big(P\texttt{;Tag}\ x\texttt{;} \Uparrow g\texttt{.}gs\big) \to \big(P'\texttt{;Tag}\ x\texttt{;} \Downarrow r\big)} \qquad \text{(tag.2)}
$$

We also need to amend the previous definitions in the operational semantics to be aware of this compound goal form. The full amended definitions are given in the appendix. For intuition, we give an example of the behaviour produced below.

The rules now contain the notation $g\texttt{.}gs$, which denotes a goal in the form of a sequence of the identifier g and the remaining goal gs. The rule (tag.1) handles the case where $x = g$, and so we continue the search for the remainder of the goal, gs, in the preceding program P. The rule (tag.2) handles the case where $x \neq g$, and so we pass the whole goal to the preceding program unmodified.

```
1  Load head <- 3;
2  Load tail <- false;
3  Tag xs;
4  Load head <- 2;
5  Assign tail <- xs;
6  Tag xs;
7  Load head <- 1;
8  Assign tail <- xs;
9  Tag xs;
10
11 Assign x <- xs.tail.tail.head;
12 Assign ys <- xs.tail;
```

```
13 Assign y <- ys.head;
```

This program above produces a data structure similar to a linked list. The expected behaviour is that finding the value of `xs.head` results in the value 1, finding the value of `xs.tail.head` results in the value 2, and finding the value of `xs.tail.tail.head` results in the value 3. The assignment to x should result in the value of 3. The assignment to `ys` gives the contents of the list of `xs` without its first element. The assignment to `y` then gives the value of 2.

An execution of the program with the goal `xs.tail.head` proceeds as follows. First, the **`Tag`** statement on line 9 is encountered. Because the tag matches the first identifier of the goal, the goal is reduced and the program continues the execution for `tail.head` in the program of lines [1-8]. Next, the **`Assign`** statement on line 8 is encountered. Because the left hand side of the assignment matches the first identifier of the goal, the right hand side `xs` is substituted into the goal, and execution continues with the goal of `xs.head` in the program of lines [1-7]. The **`Load`** statement on line 7 is skipped, and then the **`Tag`** on line 6 matches the goal and so executes with the goal of `head` in the program of lines [1-5]. Finally, the **`Assignment`** on line 5 is skipped and the **`Load`** on line 4 matches, loading the value of 2 into the variable **`head`**. This value is propagated back up and is returned as the result of the search for `xs.tail.head`.

4.2 Higher Level Constructs

The core language described in section 4.1, while complete, is not practical to write code for. All calculations have to be described step by step as separate binary and unary operations. Constants have to be manually loaded. The programmer is required to understand and use unfamiliar notions of calling, scope, and "tagging". In this section, we introduce higher level constructs which could potentially improve the programming experience. Note that while some of these higher level constructs are specific to the particular core language described above, some, such as constructs for handling infinite loops, are generally applicable to imperative lazy languages, and represent one of the main contributions of this report.

These higher level constructs are all built on top of the core language. This means that they can be implemented as a compilation step which produces a program in the core language. The benefit of this is that it keeps the size of the

core language and operational semantics small. We either specify explicitly, or give an indication of what may how such a compilation step might proceed.

4.2.1 Expressions

In the core language the only available commands for manipulating values are **`UnOp`** and **`BinOp`**, each of which executes a single unary or binary operation respectively. This is a simple approach which is good for reducing the size of the operational semantics, but requires many statements to be written to achieve even mildly complex calculations. A relatively simple extension to the language would allow complex expressions to be written in the place of a goal, allowing for statements such as `Assign x <- (y + z) * r - 5;`. A compilation step could expand these expressions into a sequence of single assignments. For example, the **`Assign`** statement shown before might expand into the following program:

```
1 BinOp temp1 <- y + z;
2 BinOp temp2 <- temp1 * r;
3 BinOp x <- temp2 - 5;
```

Formally, the expression is parsed into an expression tree (where leaf nodes are either other variables or constants, and non-leaf nodes are operators on their children. This expression tree is then serialized in into a sequence of **`UnOp`** and **`BinOp`** statements, ordered by any topological sorting of the tree from the leaves to the root. Intermediate results are held in temporary variables.

It is worth noting that this expansion of expressions is probably performed by every language with complex expressions, because CPUs typically offer primitive operations corresponding only to binary and unary operators, which must be called in sequence to evaluate expressions.

4.2.2 Linked Lists

Linked lists are introduced informally in section (4.1.12) introducing the **`Tag`** statement. The mechanism behind them is to store a head and a tail variable in the global scope, and then to tag the global scope at that position with an identifier representing the list. The head and tail can then be accessed later by specifying a goal such as `xs.tail.tail.head`.

Because this pattern is seen commonly in the examples we've considered, it makes sense to introduce a higher level syntax for the creation of a linked list. We propose a syntax of `Cons xs <- a:bs`, where `xs` is an identifier to tag the resulting link list with, and `a` and `bs` are goals (or expressions) describing the values to load into the head and the tail of the list respectively. Note that we refer to the plural `bs` because typically the tail would be another linked list, while the head would typically be a value. A compilation step could transform a `Cons` statement into a small program snippet. For example, the **`Cons`** statement shown before might expand into the following program:

```
1 Assign head <- a;
2 Assign tail <- bs;
3 Tag xs;
```

4.2.3 Arbitrary Data Structures

Similarly to **`Cons`** for linked lists, arbitrary data structures can also be represented. For example, a tree data structure might contain left and right subtrees as well as a value. Rather than explicitly **`Tag`**ging such a structure, a language could provide some facility to define custom data types and automatically produce code for creating and extracting values from them.

4.2.4 Function Definition

The existing **`Call`** function, while roughly equivalent to function calling, does not provide several niceties that are common in existing imperative languages. Namely, arguments are just inherited through the global scope, all variables modified are part of the global scope left at the end of the call, and the scoping is "dynamic" — the environment visible within a call body is that of the call site, not the definition site.

A more traditional function definition and calling mechanism can be provided by a higher level construct which compiles to a program in the core language. First, function arguments could be assigned by automatically generated code. Second, return values could be limited by wrapping a call in an **`Expose`** statement. Third, static scoping could be achieved by automatically generating a unique tag to identify the environment of the call site, and automatically prefixing all or some of the

function's values with that tag. Finally, it may even be possible to introduce a behaviour similar to "closures" or "static local" variables, where variables internal to a function are maintained across function calls.

4.2.5 Finite Loops

Finite loops, such as "for" or "while" loops in traditional imperative languages, are implemented in this language using the `Call` statement. Because they are commonly used it might be useful to programmers to have statements which give these looping behaviours directly.

The "for" statement we give has a similar behaviour to the `for` statement found in C, with an initialization, a condition, an increment, and an arbitrary body subprogram. A `For` statement might take a form like the following:

For (⟨*program*⟩ ⟨*goal*⟩ ⟨*program*⟩)
⟨*program*⟩

Where the first two programs perform initialisation and increment, the goal is tested to determine whether to continue looping, and the third program contains the body of the loop.

The "while" statement we give has a similar behaviour to the `while` statement found in C, with a condition and an arbitrary body subprogram. A `While` statement might take a form like the following:

While ⟨*goal*⟩ ⟨*program*⟩

Where the goal is tested to determine whether to continue looping, and the third program contains the body of the loop.

4.2.6 Infinite Data Structures

As explained in the background at (2.1.2), laziness enables creation of "infinite" data structures, where the structure is generated as it is requested. This behaviour is relatively unusual to imperative programming languages. Inspiration can be

drawn from iterators and generators, but even they are not entirely equivalent as our lazy imperative language allows for infinite trees and graphs as well as just infinite lists. We introduce two high level concepts for dealing with these infinite data structures: a "for in" loop, and a "double ended while" loop.

The "for in" loop takes linked lists as described in (4.2.2) and iterates through them, executing a program once for every element. This is a common pattern in the examples we've considered. A **`ForIn`** statement might take a form like the following:

`ForIn` ⟨*goal*⟩ ⟨*program*⟩

Where the goal finds a linked list through which to iterate.

The "double ended while" loop is a generalisation of a traditional while loop. The double ended while loop consists of a program to be executed beforehand, a condition, and a program to be executed afterwards. This in useful in the case where there is some computation that should happen every iteration before a condition is checked. In fact, the pattern which motivates this structure also exists in non-lazy imperative programming languages. For example, in the field of competitive programming, problems are often specified in which input consists of several lines terminated by a line containing a `#` character. Some code should be run before a condition (actually performing input), a condition should be checked (whether the line contains a `#`), and some code should be run after the condition, if it was true (the actual computation). Normally this requires duplication of code around a normal while loop.

This pattern occurs more frequently in a lazy language, where the program can be thought of as expanding as calls are made. By limiting to just post-conditional or pre-conditional expansion, significant expressivity is lost. A proposed syntax for the double ended while loop is the following:

`While` ⟨*goal*⟩ **`in`** ⟨*program*⟩ **`do`** ⟨*program*⟩

4.3 Errors

We have not considered errors in any great depth. If a program in the core language is not matched by any rule, then such a program will not execute. This requires implemented programs to not contain errors such as referring to unassigned variables, or dividing by zero, for any results to be given.

One approach to properly address errors would be to include a value type to represent an error in the computation of a value. This could be "caught" to handle the error and could contain a descriptive message of the cause of the error. This, however, was not a priority in the implementation or investigation.

5 Results

While our demonstration implementation is not focussed on real-time efficiency, we can still run experiments in it to extract meaningful results. As the implementation directly implements the operational semantics, we can record metrics such as the number of applications of each rule. We can also record the identifiers and goals in each rule application to investigate details of the performance of various algorithms.

5.0.1 Sorting

We implemented the quicksort algorithm to sort a list of integers. For 100 permutations of the list of integers from 1 to 10, we sorted and found each element separately. We measured the number of comparisons (the number of times the `BinOp` rule is applied with the `<` operator) for each run. The average results across all 100 permutations are given in table (5.1)

The average complexity of the quicksort algorithm is $\mathcal{O}(nlogn)$ comparisons [2]. In our case of a list of 10 integers, one would expect around 33 comparisons on average. To find the minimum element of a list (the first element of a sorted list), one would expect $\mathcal{O}(n)$ comparisons, as each element must be considered exactly once. In our case, we would expect 9 comparisons. While our measurement for $n = 1$ is not 9, it is less than the number of comparisons required to sort the entire list. This demonstrates that the full list is not being sorted, due to laziness. The general trend of increasing results also agrees with this, suggesting that the unnecessary end of the list is not being sorted. The results are in general higher than the expected theoretical results, and this is probably because of a combination of a naive quicksort implementation and a naive interpreter implementation. It could possibly by improving the caching behaviour of the interpreter.

Unfortunately, our demonstration interpreter is not efficient enough to prac-

n	Number of Comparisons
1	27.72
2	31.22
3	35.48
4	38.48
5	41.64
6	44.72
7	46.3
8	47.74
9	48.32
10	48.32

Table 5.1: Average number of comparisons when finding the nth element of a random permutation of the list [1..10]

tically run larger tests. As discussed in section (6.3), the current interpreter is inefficient in its representation of the program state, and there are approaches that could improve this.

6 Discussion

6.1 Is this Language Imperative?

At this point, a question to which the answer is not immediately obvious is: is this language actually imperative? In particular, the fact that the operational semantics describes transformations to the program is reminiscent of reduction rules in functional languages like Lisp or Haskell. As discussed in section (2.2), the distinction between imperative and functional languages is not entirely clear or objective.

One main feature used to distinguish these programming paradigms is the presence of statements or expressions. Functional languages are often described in terms of nested expressions, while imperative languages are often described in terms of sequences of statements. We have described this language in terms of sequences of statements, which is a piece of evidence suggesting that it could be considered imperative.

Another feature used to distinguish these paradigms is the presence and prevalence of state. Many functional languages strongly limit access to any sort of global state, and demand that functions be "pure" with no side-effects. The language defined here allows modification of global state, which is another piece of evidence suggesting that it could be considered imperative. It is worth noting that this distinction might be decreasing in value, as manipulating global state is generally recommended against in imperative languages, and in some languages (such as Rust) this recommendation is enforced.

Overall, there do exist arguments that this language is imperative. In any case, it is probably sufficiently different from existing functional languages to be interesting as an imperative-flavoured language.

6.2 Applications

One question beyond the primary of this research regards possible applications of imperative lazy execution. This work was not conducted with any particular application in mind, and the motivation was largely just to explore this area of programming languages.

That said, this work is potentially applicable outside of academia. We have not uncovered any performance reason which would prevent lazy execution from being used in a production imperative language. As discussed in section (2.1), laziness can provide opportunities for abstraction and can in some cases can improve performance. Laziness in an imperative programming language might be more accessible to programmers only familiar with imperative languages. However, despite the high-level constructs introduced in section (4.2), laziness might bring with it some challenges, largely in the handling of side effects but also conceptually in general.

6.3 Implementation

The implementation of the language has not been significantly discussed elsewhere in this report. The current implementations just implement the operational semantics, which is useful for generating results and producing evidence to support the validity of the semantics. They are not, however, optimised at all for performance. This was sufficient for our time-independent results, but there are several avenues to improve performance, which are worth discussing.

First, the current implementation in Haskell does not represent the program state efficiently. Haskell's native list type is a singly linked list, and so is not efficient for concatenation. Concatenating programs is required by the operational semantics of the `If`, `Call`, and `Expose` statements. This could be addressed by using an alternative data structure which allows for more efficient concatenation. For example, a doubly linked list could improve concatenation performance. Future research could investigate alternative representations.

Second, the `Expose` statement requires semantically inserting a marker in a program, to identify the point at which a concatenation occurs. In the current implementation, this marker is searched for linearly when the program needs to be split. A more advanced implementation could maintain a pointer to track the

position of the marker and split the program in constant time.

Third, the current implementation is an interpreter, meaning that a program is run by interpreting it as it is run, rather than by compiling to an optimised binary. Significant research would be required to investigate the applicability of compilation and optimisation to a lazy imperative language. There are decades of research into compiling Haskell code efficiently, some of which may be applicable here [18]. One benefit of compilation would be that the dependencies between statements could be statically analysed to reduce searching step by step through the program. For example, a statement assigning the value of some variable could potentially have a pointer to the exact location of the previous assignment to that variable.

Finally, the operational semantics does not specify any caching behaviour. There are some cases where a program will perform poorly due to many searches deep into the program. This poor performance could be improved by caching values in multiple places as they are found. There would be decisions to make in implementing this: whether there should be a cache per instruction or caches interspersed at some lower rate between instructions, how long a cache should remember a value for, or whether a cache can prioritise expensive-to-compute values in memory-restricted environments. Future work could involve researching the performance of the language with various approaches to caching.

6.4 Usability

Another question beyond the scope of this research regards the actual usability of a lazy imperative programming language. Part of the motivation of behind this work was to combine the potential benefits of lazy evaluation with the familiarity felt by many programmers towards imperative programming languages. Whether our lazy language could be easily used by programmers only familiar with imperative programming is an unanswered question and would be non-trivial to establish.

7 Conclusion

In this research project we have investigated how laziness could apply in imperative programming languages. We have produced a specification of a simple imperative programming language is lazily executed. We have also investigated and described several high level constructs which address the intuitive difficulties in using a lazily executed imperative programming language. Finally, we have demonstrated the lazy nature of our language with examples and results.

The contribution of this research is primarily in the discussion of both the core language and the high level constructs which aid in the implementation of common patterns. These are the areas where significant iteration and consideration of alternatives occurred.

We did not find any results which would suggest that laziness in imperative languages is not feasible, either in terms of implementation or soundness. As such, there is potential future work in this area, as outlined in section (6), in investigating better implementations, potential applications, and usability.

Bibliography

[1] John Daintith and Edmund Wright. A Dictionary of Computing: lazy evaluation. `http://www.oxfordreference.com/10.1093/acref/9780199234004.001.0001/acref-9780199234004-e-2850`.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[3] Heinrich Apfelmus. Modular code and lazy evaluation in haskell. `https://hackhands.com/modular-code-lazy-evaluation-haskell/`. Accessed: 2017-10-15.

[4] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[5] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real world haskell: Code you can believe in*. " O'Reilly Media, Inc.", 2008.

[6] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.

[7] Mary Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, 1(4):10–26, Oct 1984.

[8] Peter Van Roy. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:9–47, 2009.

[9] Howell Jordan, Goetz Botterweck, John Noll, Andrew Butterfield, and Rem Collier. A feature model of actor, agent, functional, object, and procedural

programming languages. *Science of Computer Programming*, 98, Part 2:120 – 139, 2015. Special Issue on Programming Based on Actors, Agents and Decentralized Control.

[10] Working draft, standard for programming language c++, n3337. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf`. Accessed: 2017-10-15.

[11] Jeremiah Willcock, Jaakko Järvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine. Lambda expressions and closures for c++, n1968. Accessed: 2017-10-15.

[12] Functional Programming Modules. `https://docs.python.org/3/library/functional.html`. Accessed: 2017-10-15.

[13] The Rust Reference. `https://doc.rust-lang.org/reference/`. Accessed: 2017-10-15.

[14] David A. Turner. Some history of functional programming languages. In *Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829*, TFP 2012, pages 1–20, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[15] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[16] Hans Hüttel. Transitions and trees. *Cambridge University Press*, 2010.

[17] Hanne Riis Neilson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Weily, 1991.

[18] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

A Full Language Description

A.1 Grammar

⟨*identifier*⟩ ::= ⟨*lowercase*⟩{⟨*alphanumeric*⟩}

⟨*goal*⟩ ::= {⟨*identifier*⟩`.`}⟨*identifier*⟩

⟨*value*⟩ ::= ⟨*boolean*⟩
| ⟨*number*⟩
| ⟨*program*⟩

⟨*binary-operator*⟩ ::= `+` | `-` | `*` | `/` | `%`

⟨*unary-operator*⟩ ::= `!`

⟨*statement*⟩ ::= `Load` ⟨*identifier*⟩ `<-` ⟨*value*⟩`;`
| `Assign` ⟨*identifier*⟩ `<-` ⟨*goal*⟩`;`
| `BinOp` ⟨*identifier*⟩ `<-` ⟨*goal*⟩ ⟨*binary-operator*⟩ ⟨*goal*⟩`;`
| `UnOp` ⟨*identifier*⟩ `<-` ⟨*unary-operator*⟩ ⟨*goal*⟩`;`
| `If` ⟨*goal*⟩ `then` ⟨*program*⟩`;`
| `Call` ⟨*goal*⟩`;`
| `Tag` ⟨*identifier*⟩`;`
| `Expose` ⟨*identifier*⟩ `in` ⟨*program*⟩`;`
‖ `Marker;`

⟨*program*⟩ ::= {⟨*statement*⟩}

A.2 Reduction Rules

$$\frac{x = g}{\big(P\texttt{; Load } x \texttt{ <- } v\texttt{; } \Uparrow g\big) \to \big(P\texttt{; Load } x \texttt{ <- } v\texttt{; } \Downarrow v\big)} \quad \text{(load.1)}$$

$$\frac{x \neq g \qquad \big(P \Uparrow g\texttt{.}gs\big) \to \big(P' \Downarrow r\big)}{\big(P\texttt{; Load } x \texttt{ <- } v\texttt{; } \Uparrow g\texttt{.}gs\big) \to \big(P'\texttt{; Load } x \texttt{ <- } v\texttt{; } \Downarrow r\big)} \quad \text{(load.2)}$$

$$\frac{x = g \qquad \big(P \Uparrow ys\texttt{.}gs\big) \to \big(P' \Downarrow r\big)}{\big(P\texttt{; Assign } x \texttt{ <- } ys\texttt{; } \Uparrow g\texttt{.}gs\big) \to \big(P'\texttt{; Assign } x \texttt{ <- } ys\texttt{; } \Downarrow r\big)} \quad \text{(assign.1)}$$

$$\frac{x \neq g \qquad \big(P \Uparrow g\texttt{.}gs\big) \to \big(P' \Downarrow r\big)}{\big(P\texttt{; Assign } x \texttt{ <- } ys\texttt{; } \Uparrow g\texttt{.}gs\big) \to \big(P'\texttt{; Assign } x \texttt{ <- } ys\texttt{; } \Downarrow r\big)} \quad \text{(assign.2)}$$

$$\frac{x = g \qquad \big(P \Uparrow ys\big) \to \big(P' \Downarrow y\big) \qquad r = opy}{\big(P\texttt{; UnOp } x \texttt{ <- } op\ ys\texttt{; } \Uparrow g\texttt{.}gs\big) \to \big(P\texttt{; Load } x \texttt{ <- } r\texttt{; } \Downarrow r\big)} \quad \text{(unop.1)}$$

$$\frac{x \neq g \qquad \big(P \Uparrow g\texttt{.}gs\big) \to \big(P' \Downarrow r\big)}{\big(P\texttt{; UnOp } x \texttt{ <- } op\ ys\texttt{; } \Uparrow g\texttt{.}gs\big) \to \big(P'\texttt{; UnOp } x \texttt{ <- } op\ ys\texttt{; } \Downarrow r\big)} \quad \text{(unop.2)}$$

$$\frac{x = g \quad \big(P \Uparrow ys\big) \to \big(P' \Downarrow y\big) \quad \big(P'' \Uparrow zs\big) \to \big(P' \Downarrow z\big) \quad r = y\ op\ z}{\big(P\texttt{; BinOp } x \texttt{ <- } ys\ op\ zs\texttt{; } \Uparrow g\big) \to \big(P''\texttt{; Load } x \texttt{ <- } r\texttt{; } \Downarrow r\big)} \quad \text{(binop.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\,.\,gs\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{; BinOp } x \texttt{ <- } ys\ op\ zs\texttt{; } \Uparrow g\,.\,gs\Big) \to \Big(P'\texttt{; BinOp } x \texttt{ <- } ys\ op\ zs\texttt{; } \Downarrow r\Big)} \quad \text{(binop.2)}$$

$$\frac{\Big(P \Uparrow ys\Big) \to \Big(P' \Downarrow \texttt{true}\Big) \quad \Big(P'\texttt{; } Q \Uparrow gs\Big) \to \Big(P'' \Downarrow r\Big)}{\Big(P\texttt{; If } ys \texttt{ then } Q\texttt{; } \Uparrow gs\Big) \to \Big(P'' \Downarrow r\Big)} \quad \text{(conditional.1)}$$

$$\frac{\Big(P \Uparrow ys\Big) \to \Big(P' \Downarrow \texttt{false}\Big) \quad \Big(P' \Uparrow gs\Big) \to \Big(P'' \Downarrow r\Big)}{\Big(P\texttt{; If } ys \texttt{ then } Q\texttt{; } \Uparrow gs\Big) \to \Big(P'' \Downarrow r\Big)} \quad \text{(conditional.2)}$$

$$\frac{\Big(P \Uparrow fs\Big) \to \Big(P' \Downarrow R\Big) \quad \Big(P'\texttt{; } R \Uparrow gs\Big) \to \Big(P'' \Downarrow r\Big)}{\Big(P\texttt{; Call } fs\texttt{; } \Uparrow gs\Big) \to \Big(P'' \Downarrow r\Big)} \quad \text{(call)}$$

$$\frac{x = g \quad \textbf{unique marker } m \quad \Big(P\texttt{; } m\texttt{; } Q \Uparrow g\,.\,gs\Big) \to \Big(P'\texttt{; } m\texttt{; } Q' \Downarrow r\Big)}{\Big(P\texttt{; Expose } x \texttt{ in } Q\texttt{; } \Uparrow g\,.\,gs\Big) \to \Big(P'\texttt{; Expose } x \texttt{ in } Q'\texttt{; } \Downarrow r\Big)} \quad \text{(expose.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\,.\,gs\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{; Expose } x \texttt{ in } Q\texttt{; } \Uparrow g\,.\,gs\Big) \to \Big(P'\texttt{; Expose } x \texttt{ in } Q\texttt{; } \Downarrow r\Big)} \quad \text{(expose.2)}$$

$$\frac{\Big(P \Uparrow gs\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{; Marker; } \Uparrow gs\Big) \to \Big(P'\texttt{; Marker; } \Downarrow r\Big)} \quad \text{(marker)}$$

$$\frac{x = g \qquad \Big(P \Uparrow gs\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{; Tag } x\texttt{; } \Uparrow g\texttt{.}gs\Big) \to \Big(P'\texttt{; Tag } x\texttt{; } \Downarrow r\Big)} \quad \text{(tag.1)}$$

$$\frac{x \neq g \qquad \Big(P \Uparrow g\texttt{.}gs\Big) \to \Big(P' \Downarrow r\Big)}{\Big(P\texttt{; Tag } x\texttt{; } \Uparrow g\texttt{.}gs\Big) \to \Big(P'\texttt{; Tag } x\texttt{; } \Downarrow r\Big)} \quad \text{(tag.2)}$$