Lambda Calculus with Lifetimes and Higher Kinded Types

Final Year Project Report

Felix Bowman
Candidate Number: 122587

Supervisor: Dr. Martin Berger

University of Sussex
Department of Informatics

Contents

Listings

Declaration

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Felix Bowman

Chapter 1

Introduction

1.1 Project Aims

The purpose of this project is to design a type system for a programming language that incorporates Higher Kinded Types with a system that guarantees memory safety at compile time. Higher Kinded types are necessary to neatly express certain desirable programming features. Rust is a programming language that provides memory safety through compile-time checks. This aspect of Rust is known as the *ownership* model, of which *lifetimes* and *borrowing* are key concepts [15]. This project attempts to implement a type checker for this language as a Haskell program, and investigate how higher kinded types and the Rust ownership model interact.

1.2 Safety in Programming Languages

As programs become larger and more complex, methods have been devised to detect and mitigate classes of errors. Ideally, all errors would be detected before a program is run. To this end, type system have been developed that either eliminate certain classes of errors or make certain styles of bug-prone programming hard or impossible. While doing this, it is important that a type system still benefits the programmer by not getting in the way of expressing ideas.

1.2.1 Simple Types

Type systems are designed to help programmers reason about behaviours in the programs that they write. Types restrict the kinds of statements that may be expressed, the most obvious reason for doing this is to detect certain types of errors before a program is executed. Small, obvious errors, found in expressions such as `"this sentence" * 69` are easily detected, because multiplication makes no sense when applied to a strings of characters and a number. However, modern type systems are capable of doing far more.

1.2.2 Polymorphism

As programs become larger and more complex, separating parts of programs that do not need any knowledge of each other becomes more important. It is much easier to maintain a program composed of separate modules that have no knowledge of the inner workings of one another. Modules may interact through well defined interfaces and in this way may be swapped out or replaced much more easily. One may re-write code that conforms to an existing interface without knowing about another module's implementation details. A type system can be used to define this interface and to ensure that new code behaves as expected. Polymorphism, discussed in detail in Chapter 3, is one way of allowing more flexibility in these interfaces between parts of a program while retaining correctness.

1.2.3 Memory Safety

Some programming language features may introduce whole new families of errors. Allowing a programmer to manually allocate and deallocate space for storing objects in memory is known to be a common source of bugs. Languages like C have a reputation for being difficult to write bug free programs in because of manual memory management. Memory leaks may occur if a programmer requests some storage in memory but does not deallocate it. This can lead to programs consuming unreasonably large amounts of memory. Referencing some location in memory after it has been deallocated (and possibly allocated with some new object) is another common pitfall associated with manual memory management. Garbage collection, or the automatic allocation and deallocation of objects in heap memory, may alleviate some of these issues. However, it can incur a sometimes unacceptable runtime overhead especially in software systems where responsiveness is key.

Types can also play a role in preventing these kinds of errors. Restricting where a programmer can request or reference some location in memory allows a program to be statically analysed before it is run, and any sections of code that could cause memory related issues are shown as compile time errors. A type system can form the basis of this restriction, as shown in Section 4.5.1

1.3 The Lambda Calculus

In studying these type systems it is often useful formalize them in a system with well known and well understood behaviour. In this case the system is the lambda calculus, which functions as a miniature programming language and as a formal model for understanding the behaviour of types. The lambda calculus and how it pertains to this project is described in Section 5.1. Higher Kinded Types, or higher order polymorphism, have been well described in System F ω, itself a typed lambda calculus [12].

1.4 Goals of This Project

A parser, type checker, and simple interpreter will all be implemented along with a test programs demonstrating some intended behaviours. The design of the language is described in Section 5. Haskell will be used as the language of implementation, and is described in Chapter 6 along with the reasons for choosing it. Some of the main goals of this project are:

- Developing a very simple programming language that incorporates dynamic memory allocation, modelling functions that operate on a heap.
- A system restricting the allocation, use, and deallocation of memory, encoded into the type system. This aspect is based on the type system of the Rust programming language, which guarantees memory safety through associating memory tied to function scope.
- A type system that allows for higher order polymorphism, described in Chapter 3. Formalised in System F ω, it allows for a neat implementation of very generic, high level programming concepts such as Monads and Functors.

The notion of an addressable area of memory, as well as mechanisms for allocating and deallocating areas of that memory, will be added to the base lambda calculus. The implementation of the type system, as well as the parser, build system, test framework, and interpreter are described in Chapter 6. A formal description of the language can be found in Chapter 5.

1.5 Project Overview

The rest of this report is divided in several chapters. Chapter 2 lists the ethical considerations of this project. Chapter 3 introduces polymorphism before introducing Higher Kinded Types. Memory allocation and the problems associated

with it is introduced in Chapter 4. Chapter 5 then introduces the lambda calculus as way of reasoning about type systems. The design of the language is introduced by building up features from existing languages and then incorporating them into one final grammar. Software engineering issues are tackled in Chapter 6, including the test framework and build system used. Key tests demonstrating desired behaviour are given in Chapter 7. Issues about the effectiveness of this type system are described in Chapter **??**. Finally, unmet requirements and further work to be done is covered in Chapter 8.

Chapter 2

Professional and Ethical Considerations

No part of this project requires human participation and as such there are no ethical considerations.

Chapter 3

Parametric Polymorphism

This section introduces Parametric Polymorphism, which is found in many programming languages. It is used to make programming languages more expressive while attempting to prevent errors from occurring at runtime. Polymorphism is one way of sticking to a core programming principle: Functionality in a program should only be implemented in one place in the source code. If different functions are meant to do very similar things, it is generally beneficial to combine them into one by parametrizing sections that differ. This allows code to be updated, maintained, and deleted more easily and so becomes very necessary in large codebases. This parametrizing of areas of code is known as abstraction. Abstraction at the levels of values is very common in the form of functions; parametric polymorphism allows for the abstraction at the level of types.

The rest of this chapter attempts to explain how polymorphism is useful, and how it is used in languages like Java to reduce code duplication and increase expressiveness. Generics, or parametric polymorphism in Java, is used extensively in Java's collections framework, which provides very general data structures for organizing data in programs. These structures are practical to implement and use partly because of Generics.

Finally, Functors are introduced as an concept that we would like to be able to represent and use in a programming language, but cannot do so in languages with the expressive power of Java. Higher Kinded Types are introduced to provide very generic solutions to programming problems, and form the basis of a large part of this project.

3.1 A World Without Generics

In software engineering one of the principles to keep in mind when attempting to write software is Don't Repeat Yourself, known as the DRY principle. If lots of similar behaviour exists in different places in a codebase, then a change to this behaviour will result in the code being modified in several places. Such changes to behaviour are common; and not applying changes to one area while applying them to another area can be a source of bugs. There are other reasons not repeat oneself in a codebase as well. Less time is spent writing code if a function only has to be written once, and lots of similar code can distract from the meaning of a program.

As an example, consider writing a list class to store some strings in a language like Java. The class may have an implementation that looks something like this:

```
class StringList {

    public boolean add(String e) {
        // implementation...
    }

    public String getAt(int index) {
        // Get the string at position index...
    }

    public int size() {
    }
```

```
    // other sensible methods that you might want in a list...
}
```

Listing 3.1: String List in Java

This seems all very well and good. However, at a different part of the program we might want to have a list of `Booleans`:

```
class BoolList {

    public boolean add(Boolean e) {
        // add a boolean to the list
    }

    public Boolean getAt(int index) {
        // Get the bool at position index...
    }

    public int size() {
    }
}
```

Listing 3.2: A very similar class

The similarity between these classes is obvious: the methods that they implement are very similar except for the types in method signatures. `BoolList`'s `getAt` methods returns a `Boolean`, and `StringList`'s `getAt` returns a `String`. It is easy to see that the implementation for both of the `getAt` methods would be very similar. Now imagine if there was a bug in the implementation of this method. This bug would likely be present in both of the implementations and so a programmer would have to fix both of these implementations. It would be very possible for the programmer to forget or ignore one of these methods, leading to inconsistent behavior in the program. Even if the programmer did remember to change the codebase in two places, there would still be twice as much code to change. This becomes more of a problem for every new list type that the programmer wishes to implement.

One might be tempted to leverage Java's inheritance to solve this issue. A general list class that stores Java's `Object` class might be the answer. After all, `String` and `Boolean` both extend `Object`. At first glance this seems like a sensible solution, any changes to how lists would only have to be put into effect in this one class. This would be a vast improvement on making changes to every type of list that we implement. The class might look something like:

```
class List {

    public boolean add(Object e) {
        // add whatever you like to the list!
    }

    public Object getAt(int index) {
        // Get the whatever object is stored at index
    }

    public int size() {
    }
}
```

Listing 3.3: A more general approach.

This class looks very similar to the types of list that were implemented in Listings 3.2 and 3.1. However, in the signatures of the methods we use `Object` so we can put whatever we like in the list. The class could be used like this:

```
List list = new List();
list.add("This seems to work well!");
String firstElem = (String) list.getAt(0);
```

However, there could be a problem here. On the last line we get an element from the list, using the `getAt` method, which returns an `Object`. An instance of type `Object` is not very useful because we probably want to call some class-specific methods on our element that we pulled from the list. The example above gets around this by using casting.

Casting is a way of telling the Java compiler that one type should be treated as another. Specifically it is often used to tell the compiler that the class that is being casted from is actually a subtype of the class that is being casted to. In this case it is used to tell the Java compiler that the instance of `Object` that is being returned from the `getAt` method is actually a `String`. This is done so that we can access all of the methods that the `String` class provides, and so this object can be used where ever an instance of the `String` class is expected.

Casting from `Object` cannot be checked by the compiler. This snippet shows what could potentially go wrong in this situation:

```
List list = new List();
list.add(new Boolean(true));
String firstElem = (String) list.getAt(0);      // This line will compile
```

The above code compiles but will throw a `ClassCastException` when run because String is not a subclass of Boolean. The Java compiler cannot tell from the above that the list object is holding `Boolean` values, so cannot warn the programmer that the above code does not make sense. Any safety provided by the type system has been subverted and the code may be in production somewhere before the obvious error is detected by a human, or worse, throws a runtime exception. This is the kind of problem that Java's Generics were intended to solve.

3.2 Parametrizing Constructors

Just as methods in Java may have formal parameters that abstract values out of the method body, Generics allow types to be abstracted out of the implementation of a class. When a class is constructed a type can be passed in as an argument which can be referenced by the methods that the class provides. The type that is passed into a class is referenced with a type variable, just is arguments to methods are referenced by variables in the scope of that method. This has the benefit that code can be type-checked at compile time. Casting, as demonstrated in Section 3.1, can lead to run time errors which can be difficult to find before they cause errors.

With Generics the list example from the previous section can be implemented like this:

```
class List<T> {
    public boolean add(T e) {
        // add something of type T to the list
    }

    public T getAt(int index) {
        // Get the object of type T stored at index
    }

    public int size() {
    }
}
```

Listing 3.4: List class with Generics

Here the list is parametrized with some as yet unknown type, `T`, in the declaration `class List<T>`. When an instance of this class is created a type is passed into the constructor and the instance becomes a new, specific type of list holding objects of that type. This list can be used like this:

```
List<String> list = new List<String>();
list.add("Much better!");
String firstElem = list.getAt(0); // No casting!
```

The `String` type is passed into the `List` type constructor here. The `T` type variable in Listing 3.4 is instantiated with the `String` type and methods like `getAt` return a `String`. There is no need for casting and the type correctness of this code can be checked at compile time. If we tried to assign the result of the `getAt` method to a boolean variable here the program would not compile.

3.3 More Generic Containers

Any class that acts as a container for other types of object benefits from being parametrized with the type of object that it holds. In the list example earlier in the chapter it did not matter what kind of element the list was meant to store, all that mattered was that there was a list of them. Most data structures are this general, in that the type of element inside the structure does not matter. This is where parametric polymorphism is most useful.

Binary trees are another example of a data structure that can store any type of object. A node in a binary tree can either be empty, or hold an element and a pointer to two other nodes. The element that is held in that tree can be of any type– a perfect opportunity to use generics. The implementation could look something like this:

```
class Tree<T> {
    private Tree<T> left;       // pointer to the left branch
    private Tree<T> right;      // pointer to the right branch
    private T elem;             // Data that this node stores

    public void insert(T e) {
        // insert element into the tree
    }

    public boolean contains(T e) {
        // is e in this tree?
    }

    public int size() {
    }
}
```

Listing 3.5: Binary tree with generics

Very similarly to the list implementation, the tree class must be instantiated with some type, referred to as `T` in the body of the class. Both of these data structures act as generic containers.

This kind of abstraction, where types are parametrized in the body of classes, is known as first-order polymorphism. This notion is formalized in language called System F, described in Chapter 5. However this report attempts to explain higher-order polymorphism, a concept that cannot be expressed in languages like Java. In order to understand why higher order polymorphism is useful and why it cannot be expressed in Java it is important to understand some additional concepts.

3.4 `map` and Higher Order Functions

As mentioned, both the list in Listing 3.4 and the tree in Listing 3.5 can be thought of as containers that hold some type of object. It is very common in programs to apply a function to every element in collection. In Java, this is commonly expressed with a for-each loop. For example, to get a new list of integers by adding one to every element from an old list of integers one would write:

```
List<Integer> intList = new ArrayList<>();
intList.add(1);
intList.add(2);
intList.add(3);

List<Integer> outList = new ArrayList<>();

for (Integer i : integerList) {
    outList.add(addOne(i));         // outList will contain [2, 3, 4]
}
```

Manipulating elements of a collection in this way is very common. However, all we are doing here is applying a function to every element in this container. It would be nice if one could apply the same kind of transformation to the binary tree that we defined, without worrying details of tree traversal in this section of program.

In functional programming this problem is solved by a specific higher-order function, commonly called `map`. A higher-order function one that takes another function as a parameter. `map` as defined for our list class would take a function as a parameter and apply that function to every element in a list. The example from above could be written as:

```
List<Integer> intList = new ArrayList<>();
intList.add(1);
intList.add(2);
intList.add(3);

List<Integer> outList = intList
    .stream()
    .map((x) -> addOne(x));     // Define an lambda function to pass to map
```

Here a lambda function is declared using Java 8 lambda syntax. The function `(x) -> addOne(x)` is declared as a without a name and passed as an argument to the higher-order function, `map`. The lambda function is then applied to every element of the list and a new list is returned.

The map function can be declared for any class that acts as a container holding other values. It takes a function as a parameter and applies this function to every the contents of that container. It would make sense to have a map function for both lists and trees. It would also make sense to try define an interface for containers that can be mapped over in this manner.

3.5 `Functor` and Mapping over Containers

Having an interface for containers that can be mapped over would be useful. Any time we wanted to apply a function to every element in a container we could pass a transforming function to the `map` function which the interface would provide and not have to worry about the implementation details of iterating over every single element. Even if the type of collection were to change, e.g. using a tree instead of a list, the transformation would hold because both of the classes would implement the interface. This concept of a mappable container is typically called Functor. We can try and write an interface for this concept in Java:

```
interface Functor<A> {
    Functor<B> map(Function<A, B> f);
}
```

Listing 3.6: An attempt to define Functor in Java.

At first glance this interface seems to be acceptable. The interface takes a type parameter, `A`, which is the type of the element that the container holds. One method is defined, `map`, which takes a function `f` as a parameter. `f` has a generic type; it takes an object of type `A` and returns an object of type `B`. `map` itself returns an object than implements `Functor` with elements of type `B`.

However the return type of `map` is not sufficient. It may return an object of any class that implements `Functor`, which will have the correct type of elements `B`, but will not be more specific than that. Casting would be needed to use the collection again:

```
List<Integer> intList = new ArrayList<>();
intList.add(1);
intList.add(2);
intList.add(3);

// Before casting, the result of map is just Functor<Integer>
List<Integer> outList = (List<Integer>) intList.map((x) -> addOne(x));
```

This results in a similar situation to the one described in the example of the list that held only `Object` in Listing 3.3. Casting is needed to use the return value of `map` as a list, so is necessary if we wanted to use any list specific methods or pass it into a function that expects a list as an argument. Again, casting is not an ideal solution because it is susceptible to run time exceptions. In other words, this Functor interface cannot be used in a type safe manner.

In order to define a generic Functor interface, where the return type of `map` is the same as the class that implements functor, we need to reference that class that implements the interface inside the interface itself. In other words, `map`

must know that it returns a `List` when the list class implements Functor, or a `Tree` when the tree class implements Functor. Hypothetically, this interface might look something like:

```
interface Functor<F<A>> {
    F<B> map(Function<A, B> f);
}
```

Listing 3.7: Hypothetical type safe functor in Java

Note that this is not valid Java. In the type parameter to this version of the interface we define a generic class constructor, `F`, that itself take a single generic type parameter, `A`. The map function then returns the *same* generic class, but parametrized with the result type `B`. In this manner we could constrain the map function to return the same instance of Functor that it is called from.

3.6 Higher Kinded Types

In order to express concepts like the Functor described in Listing 3.7 higher order polymorphism is needed. This means that we need the ability to pass classes that themselves have type parameters, like `List`, as type arguments in classes. Type constructors can be represented by type variables in the body of classes. This can be seen in the hypothetical Java functor example above, where the polymorphic type constructor is represented by the type variable `F`. If Java allowed for type constructors to be represented as type variables then the Functor syntax above might be one way of representing this.

Other generic concepts could be represented as well. Monads, a way of associating data types with specific computations, are well represented with higher order polymorphism.

Chapter 4

Memory Management

Part of this project is designing a compile time system for memory management. Before describing how this system should work, an overview of existing memory management techniques is given. Most programming languages have the ability to dynamically allocate memory on the *Heap*, which is a very useful language feature. However, the allocation, dereferencing, and freeing of heap memory is problematic and this chapter will describe some of the issues with heap allocated memory.

4.1 Memory Usage and Computational Effects

Most practical programming languages allow for storing and referencing areas in memory. When an expression is evaluated, it may have an effect on memory as well as returning a result. For example, an assignment statement in a programming does not have any obvious result, but instead operates on values in an area of memory that is accessible to other parts of the program. A language that allows for these sorts of computational effects is referred to as *impure* and one of the goals for this project is to make these features safer to use.

4.1.1 Stack and Heap Memory

There are usually two kinds of memory that may be used to store values: stack memory and heap memory. Stack memory is commonly used to store values for function calls [2]. A function is called with parameters referred to by variables, which are in scope for the duration of the function call. These local variables hold the result of evaluating some expression, which is typically stored in stack memory. This has the advantage that memory is very predictably allocated and deallocated, and this operation of allocating and deallocating can be accomplished very quickly.

However, dynamic memory allocation is sometimes needed when multiple functions need to refer to or mutate the same value; such as is the case with global variables. Typically dynamic memory allocation refers to memory management on the heap.

These values live longer than any individual scope and therefore it does not make sense to store them on the stack. Generally, heap memory is used when some value needs to be alive for a longer than a single function's execution. However this memory cannot be predictably allocated and deallocated. Programming languages have several strategies for managing heap memory.

Basic Operations

The basic operations needed in a system with memory allocation are:

Allocation

Allocation refers to creating an area in memory to store the result of an expression. This is commonly seen is

the form of variable assignment, for example `x := ref 2 + 3;`. Here the expression `2 + 3` is evaluated [1] and the result is stored in the area of memory referred to by the variable `x`. The value referred to by `x` can then be used at some later point in the program. Variables may also be reassigned, for example `y := x`. `y` would then point to the same location that `x` points to.

Dereference

Once an area of memory has been allocated, and is pointed to by a variable, some way of using the expression stored at that location is needed. Dereferencing refers to getting the expression stored at a location in memory. If we wanted to use `x` at some point after the expression `x := ref 2 + 3` had been evaluated, we could write `!x`. The result of that expression would be 5, because `2 + 3` has been evaluated and the result stored in the area of memory that is pointed to by `x`. The dereference operator, `!E`, means "go to the location in memory, `E`, and get the value stored there".

Deallocation

Deallocation refers to freeing memory once it no longer needed. It is common to not want to use memory for the whole lifetime of a program. If memory was never freed programs would use far more memory than they needed, which would not be practical. In order to solve this problem memory may be deallocated once a variable goes out of scope, which is common for stack memory, or we could introduce an explicit deallocate operator, called `free`. This operator would take a memory location and mark that location as unused so that a future allocation could reuse the same space.

4.2 Problems with Dynamic Memory Allocation

Although dynamic memory allocation is a very useful language feature it is also a notorious source of bugs. Compared to stack memory allocation, there is no enforced regular pattern of allocation and deallocation. Either the programmer must explicitly state when to allocated and deallocate memory, or an automatic system is used to management memory, called a garbage collector. If heap memory is manually managed, such as is the case in languages like C, there are some common bugs, which are described in this section.

4.2.1 Dangling References

Deallocating storage may lead to dangling references, where some reference exists that points to memory that has been deallocated. This is nearly always unintentional, however it is an easy for a programmer to make this mistake when writing a program. Using the value of deallocated storage can lead to nefarious bugs and most languages consider using deallocated storage to be undefined behaviour.

```
int *dangle()
{
    int i = 69;
    return &i; // Function return address of local variable!
}

int main()
{
    int *p;
    p = dangle();
}
```

Listing 4.1: A C program that leaves p pointing to deallocated memory.

In this example the function `dangle()` returns the address of variable that is deallocated when the function terminates. This means that part of the program that attempts to use the value returned by `dangle()` will result in result in undefined behaviour.

[1] In languages with *strict* evaluation, the expression is evaluated before the result is bound to the variable. This is not the case in all programming languages.

4.2.2 Memory Leaks

Memory leaks occur when heap storage that is no longer usable is not deallocated. This can lead to software allocating more memory for itself and in the worst case can lead to a program consuming all available memory.

```
#include <stdlib.h>

void allocate()
{
    int* a = malloc(sizeof(int) * 100); // Allocate memory on the heap
    // Return and forget about the allocated memory!
}

int main()
{
    allocate();
    // ...
    // do more stuff, while heap space that allocate used is unreachable
}
```

Listing 4.2: A C program which leaks memory.

In this C program memory is allocated in the function `allocate()`. The C library function `malloc(size_t size)` allocates `size` bytes on the heap and returns a pointer that area. However, in this example, the pointer that is returned, bound to the variable `a`, is not used and never returned. Therefore the memory has been allocated on the heap but there is no way of referencing it once the function `allocate()` goes out of scope. If the function were used in a loop, heap memory would very quickly be used up pointlessly!

4.2.3 Double Free

Once a programmer no longer needs some dynamically allocated memory they should deallocate it so the memory space is not wasted. This is can be accomplished by calling a provided library function, such as C's `free(void *ptr)`. This function takes an address of some space in heap memory as a parameter and makes it allocatable again. However, if the same address in memory were to be freed twice, then some other object that may now reside in that space may be prematurely destroyed. For example:

```
#include <stdlib.h>

int main(void)
{
    char *p, *q;
    p = malloc(100);    // allocate some heap storage
    // do some work with p...
    free(p);            // Done with p now, so we free it
    q = malloc(100);    // Allocate a new object on the heap
    free(p);            // Whoops! Deallocate p's area in storage again,
                        // potentially destroying q
}
```

Listing 4.3: Mistakenly freeing the same memory more than once.

In this example space is allocated which is pointed to by `p`. Once this storage is free, space is allocated for `q`. The space allocated for `q` might be the same space that was originally allocated for `p`. When `p` is freed again, `q` might be accidentally destroyed. Any future attempt to use `q` again after this would result in undefined behaviour.

4.3 Potential Solutions

Several design patterns and technologies have been developed for tackling the dynamic memory management issues described in Section 4.2. This section attempts to describe some of these techniques, while noting why they may not an ideal solution.

4.3.1 Garbage Collection

Garbage refers to heap allocated storage that is no longer reachable by the programmer. This memory can no longer be used and should be deallocated. It is important to note that garbage collection is not performed by the compiler, but by a runtime system [2]. Languages with garbage collection do not have manual deallocation of memory, but instead rely on a system that attempts to track when objects are no longer reachable. The programmer therefore does not have to manually annotate the program and tell the compiler when to release memory, i.e. constructs like C's `free` are not needed. Garbage collection effectively solves problems related dangling pointers, double free bugs, and certain kinds of memory leak.

Many modern language rely on garbage collection for memory management. Because of improvements in safety and ease of use Java, Python, Scala, Haskell, and many others all rely on garbage collection. This has a notable effect on programmer efficiency [8]. However, use of a garbage collector does not come without cost. Some issues are:

- A programmer can have some insight as to where objects should be freed in a program and may know how long an object should exist for. The garbage collector may not be able to make this insight. As a result a program written in a garbage collected language can use far more memory than than a program in a language with manual memory management written by an expert.
- Significant computational overhead is needed in a program because garbage collectors must always account for the worst case scenario. Because they need to be running at the same time as the program that they manage they can significantly decrease performance [8]. In lots of applications the speed of execution does not matter, but for certain types of applications the detriment to performance is unacceptable.
- Program execution may be interrupted in order to collected unused memory. The programmer cannot predict when these pauses may happen. Certain types of real-time or interactive applications should not be interrupted, so cannot be programmed in garbage collected languages.

Because of these issues, use of a garbage collected language is not always the correct decision.

4.3.2 Resource Acquisition is Initialization

Resource Acquisition Is Initialization, or RAII, is a technique, not a technology, which connects the lifetime of a resource, such as heap memory, to the lifetime of an object [13]. It is a way of managing dynamically allocated memory in languages with object oriented features.

Lifetimes

The lifetime of an object or resource refers to the length of time from when an object is initialized to when it is deallocated. It is used in object oriented programming to refer to the length of time that between an object's creation and destruction.

Lifetimes can be used for resource management, which is the essence of of the RAII idiom. Lifetimes refer to the duration that an object is usable for. Resources are allocated when an object is created, and deallocated when an object is destructed. An object's lifetime may be tied to its enclosing scope, and in this way resources may be freed when an object goes out of scope. Resources will be freed even the case of exceptions causing a function to exit early. In this manner resources will always be deallocated when an object goes out of scope, effectively solving dangling issues related to dangling pointers [15]. However, it is still a problem if these ideas are not enforced by the programming language as part of the type system. Programmers may forget to implement these ideas or may not be aware of them. However, some of these ideas can be statically checked by a type checker.

4.4 Enforcing Memory Safety in the Type System

The idea that all resources have an owner is key. This is the core concept from 4.3.2: if a resource's existence is tied to an scoped variable then the resource can be deallocated when that variable goes out of scope. If any given resource

is associated with a *single* variable then there is no way of loosing track of it. In order for this system to work them no memory must be aliasable, which is should not be because all resources are tied to a single owner.

4.4.1 Ownership and Affine Type Systems

Affine types are a development of Linear type systems. Linear type systems ensure that every variable is used exactly once. [10] In an affine type system, a variable may be used at most once, i.e. zero or one times. This has applications for managing resources in a program that do not persist the whole time that the program is running, e.g. file handles or heap allocated memory [11] [16].

4.5 Introducing Rust

4.5.1 Ownership

Values in Rust are bound to a single *owner*, which is the variable to which it is bound. When the owning variable goes out of scope, the value is freed. For example:

```
{
    let x = 69;
}
```

When `x` goes out of scope the value associated with it will also be released from memory. Because there can only be a single owner associated with a value, aliasing a variable is not allowed. This prevents any two parts of a Rust program from accessing a section of heap memory at the same time.

```
{
    let x = Thing::new();
    let y = x;
    println("()", x);
}
```

The above code does not compile because `x` has been moved to `y`, and there is an attempt to use `x` in the print statement. Using a variable as a function argument will also make it not available for use again. If a function uses some value that has been passing in as a argument it must return that same value if the scope outside that function uses it.

```
// value is comes from outside the scope of this function
fn foo(value: Bar) -> Bar {
    // do stuff with value, returning it and hence handing back ownership
    value
}
```

This is not practical, and gets even less practical once there are more arguments that must be passed back to the caller of the function. To this end Rust allows references to a value to be created. Creating a reference to a value in Rust is called borrowing. A reference to a value may be created and passed in to a function that uses it in some way, and that function does not need to thread the ownership of that variable back to the caller.

```
// instead of passing value1 and value2 in by value, create references to values
fn foo(value1: &Bar, value2: &Bar) -> u32 {
    // do stuff with value1 and value2 and return the answer
    123
}
```

There are two kinds of borrows: immutable and mutable. There may be many immutable borrows of a resource at once or a single mutable borrow. This also prevents heap data from being accessed at the same time at different points in the program. The scope of borrow must always be shorter than that of the resource which it borrows.

This system where resources are freed after the single owner goes out of scope, as well as borrows always having shorter lifetimes than original resource mean that Rust can enforce the memory safety of programs. These concepts are built into the type system.

Chapter 5

Formal Definitions

This chapter attempts to formalize some of the concepts from Chapters 3 and 4. The lambda calculus is introduced as a model of computation and as a system for reasoning about features found in programming languages. Types are introduced to the lambda calculus as well as other more complex extensions relevant to this project. The higher order polymorphic lambda calculus, known as System F ω, is included as well as a lifetime–calculus devised for this project.

5.1 Lambda Calculus

The lambda calculus is model of computation where the only behaviour is function definition and application. It is commonly used to specify programming language features and to formalize their behaviour. The reasons for useing the lambda calculus to specify programming behaviour are:

- It is a very simple language, with very few constructs to reason about. This makes it very easy to implement an interpreter for, and very easy to implement type systems for.
- It is sufficiently powerful to express all computable functions. although it is extremely simple, it is a universal model of computation.

The lambda calculus can be view as a miniature programming language or as a system for modelling abstract behaviour. reason about type systems. The lambda calculus can be viewed as a miniature programming language and a system where strict properties can be proved.

t	::=	x	variables
	|	λx . t	function abstraction
	|	t t	function application

Figure 5.1: Grammer of the untyped lambda calculus.

5.1.1 Simple Types in the Lambda Calculus

The lambda calculus can be extended with the most simple of typing systems. In order to demonstrate how the lambda calculus can be used to reason about type systems, some additional constructs added. If Boolean literals some other concepts are added it becomes more clear how types are useful in programming languages.

Assignment, allocation, and dereferencing are all incorporated into the language. These operations follow the description given in Section 4.1.1.

Unit

We need some way to talk about operations that do not have any clear result themselves, but instead operate with side effects. The most obvious example in this case is the assignment operation, which operates on the store by inserting or modifying variable bindings. None of the types we have seen so far would make any sense to type this statement so a new type is introduced: the `Unit` type. Therefore, any operation that is executed for its side effects has type `Unit`.

Terms

t	::=	x	variables
	|	λx : T . t	function abstraction, with type annotations
	|	$true$	true literal
	|	$false$	false literal
	|	if t then t else t	if-then-else expression
	|	t t	function application

Values

v	::=	λx : T . t	lambda value
	|	$true$	true literal value
	|	$false$	false literal value

Types

T	::=	T $\rightarrow$ T	type of functions
	|	$Bool$	type of boolean literals

Figure 5.2: Lambda calculus extended with simple types and booleans.

Programs can be constructed using the language defined in Figure 5.1.2 that do not make any sense. For example:

`baddef = if` $\lambda x : Bool.x$ `then` $false$ `else` $true$

This program is constructed according to the grammar of terms. However, is not well typed according to the typing rules in Figure 5.1.2. Specifically, the type of the term found in the guard of the if-expression, $\lambda x : Bool.x$ has type $Bool \rightarrow Bool$ but has to be $Bool$ for the term to be considered well typed. It makes sense to restrict the sorts of values found in the guard of an if-then-else expression. Anything other than a boolean value in that place may indicate programmer error, which could be as simple as a spelling mistake or a fundamental misunderstanding of what they are trying to express. Type systems help catch these kind of mistakes before program execution. Specifying these rules on top of the lambda calculus can be a very informative way of looking at programming language features. The rest of this chapter describes other extensions to the lambda calculus that model some of the programming concepts investigated in this report.

5.1.2 Lambda Calculus with Dynamic Memory Allocation

In order to study a system guaranteeing memory safety, we first need a language that allows for dynamic memory allocation. This language can be built on top of the simply typed lambda calculus.

Terms

t	::=	x	variables
	|	λx : T . t	function abstraction, with type annotations
	|	$true$	true literal
	|	$false$	false literal
	|	$unit$	unit constant
	|	ref t	reference creation
	|	!t	dereference operation
	|	t := t	assignment
	|	l	heap pointer
	|	if t then t else t	if-then-else expression
	|	t t	function application

Values

v	::=	λx : T . t	lambda value
	|	$true$	true literal value
	|	$false$	false literal value
	|	l	heap pointer value

Types

T	::=	T $\rightarrow$ T	type of functions
	|	$Bool$	type of boolean literals
	|	$Unit$	unit type
	|	Ref T	type of reference cells

Figure 5.3: Lambda Calculus expanded with dynamic memory allocation

5.1.3 System F

System F is an extension of the simply typed lambda calculus [12] that allows for quantification over types as well as terms. In doing this it formalizes the notion of polymorphism in programming languages, as described in Section **??**. It is used to study implementations of polymorphism in programming languages.

As the simply typed lambda calculus allows for abstraction of terms outside of terms through function definitions, System F introduces abstractions at the level of types. The system also allows for application of type level expressions. This system can be used to reason about first order polymorphism, however more extensions to talk about higher order polymorphism. These extensions are introduced in Section 5.1.4.

Terms

t	::=	x	variables
	|	$\lambda x : \mathtt{T} \,.\, \mathtt{t}$	function abstraction, with type annotations
	|	t t	function application
	|	$\lambda X \,.\, \mathtt{t}$	type abstraction
	|	t [T]	type application

Values

v	::=	$\lambda x : \mathtt{T} \,.\, \mathtt{t}$	lambda value
	|	$\lambda X.\ \mathtt{t}$	type abstraction value

Types

T	::=	$\mathtt{T} \to \mathtt{T}$	type of functions
	|	X	type variables
	|	$\forall X.\ \mathtt{T}$	universal type

Figure 5.4: System F

Examples

System F models first order polymorphism in programming languages, such as aspect of the generics found in Java. Some examples of how System F can be used are given below:

```
id = λX.λx : X.x
boolId = id [Bool]
```

This example shows type level lambdas, where X is a type variable that has been abstracted out of the function definition. The term level lambda abstracts x which has the type X. Applying `id` to a type maybe be written as `id [Bool]`, which yields $\lambda x : Bool.x$, or the identify function over boolean values. Here X in the original definition of `id` has been instantiated with the type of Booleans.

5.1.4 System F ω

System F ω is also known as the higher order polymorphic lambda calculus. It takes concepts from the polymorphic lambda calculus, or System F, and combines these with type constructors described in Section 5.1.4. From a programming point of view, System F ω allows us to write polymorphic functions not only at the level of types, but also in type constructors.

Most usefully System F ω can be used to build type safe abstractions around traditionally impure forms of computation; typically in the form of a *monad*. Not only is higher order polymorphism very good at formalizing monads, but also very good at combining them. *Monad transformers* are a key example, but outside the scope of this paper. An example of monad transformers in use can be found in Section 6.4.6. Haskell uses the concepts to great effect.

Type Operators

Type constructors were mentioned in Section **??** and this section attempts to frame this idea within the lambda calculus. As mentioned previously, type constructors can be thought of functions from types to new types. Application and abstraction at the level of types can be modelled as a type system for the lambda calculus.

Terms

t	::=	x	variables
	|	λx : T . t	function abstraction, with type annotations
	|	t t	function application

Values

v	::=	λx : T . t	lambda value

Types

T	::=	T $\rightarrow$ T	type of functions
	|	λX :: K.T	operator lambda
	|	T T	operator application
	|	X	type variables

Kinds

K	::=	$*$	kind of concrete types
	|	K $\rightarrow$ K	kind of type operators

Figure 5.5: Type operators and Kinding

Kinds play a key role here. Just as types enforce well-formed expressions at the level of terms, Kinds are needed to check that expressions at the level of types are well formed. In other words, Kinds do not allow non-sensicle expressions like `Bool Int`, because Bool is not a type constrcutor. Concrete types, also called base types, have Kind `*`. These are the types that may be the type of terms in the lanaguage like `false`. Type constructors, like `List` have kind `* \rightarrow *`. They need to be parametrized with a type that had kind `*`.

Type operators combined with System F forms the basis of System F ω. In effect System F ω turns type constructors into first class values of the language. Kind annotations are carried over from the lambda calculus with type constructors.

Terms

`t`	::=	x	variables
	|	λx : `T . t`	function abstraction, with type annotations
	|	`t t`	function application
	|	λX :: `K . t`	type abstraction
	|	`t [T]`	type application

Values

`v`	::=	λx : `T . t`	lambda value
	|	λX :: `K . t`	type abstraction value

Types

`T`	::=	`T` $\rightarrow$ `T`	type of functions
	|	λX :: `K.T`	operator lambda
	|	`T T`	operator application
	|	X	type variables
	|	$\forall X$ :: `K . t`	universal type

Kinds

`K`	::=	$*$	kind of concrete types
	|	`K` $\rightarrow$ `K`	kind of type operators

Figure 5.6: System F ω

Chapter 6

Implementation

This chapter describes how the ideas and type system described in Chapter 5 are implemented. As stated in Chapter **??**, one of the goals for this project is to build a program that will be able to parse, type check, and evaluate a language based on the lambda calculus. The most involved phase of implementation and the focus of this project involves the type checking phase, which includes lifetime checking of pointers and references, polymorphic type checking, and checking that the language of types themselves is well formed, or kind checking.

The requirements of this project influence the choice of the language of implementation. Haskell was chosen as it is well suited to the tasks of parsing, abstract syntax tree declaration, and is in general a practical language to work with.

6.1 The Haskell Language

Haskell is a general purpose, statically typed, functional language [7]. It has several desirable features for a language implementation language. These include:

- Algebraic data types, which are very good at representing abstract syntax trees, used to describe programming languages.
- Pattern matching, which is useful for deconstructing abstract syntax trees.
- A wealth of effective parsing libraries, convening several paradigms.
- Idiomatic monadic programming, which can be used to reduce error handling and state threading boilerplate out of the logic of the program.

It also has the benefit of having higher kinded types built into the language, making it ideal for testing potential features of the implemented language. The language described in Chapter 5 has been implemented as a Haskell program. The rest of this chapter describes how this was accomplished in more specific detail.

6.2 Stack, Haskell's Build System

Stack was used as the build tool for this project [14]. This made it easy to compile and test the project as well as managing libraries. Several commands were very useful in managing the project, including `stack build` to compile all changes to the project and `stack test` which ran all of the tests for the project. The lexer and parser generator source files were not Haskell source files but instead specify how Haskell source files should be built. These tools are described in detail in later sections but Stack was used to automate the building of these tools as well. Finally, Stack provides an interface to an interactive interpreter for Haskell, GHCI. This tool was used extensively to test parts of the project as they were being developed.

6.3 Parsing

A lexer generator, Alex [1], and parser generator, Happy [6], were used in this project. Using these tools in combination made parsing the language into the Haskell representation of the abstract syntax simple. The grammars provided for the parser generator are a very close approximation of Haskell data type. Both Alex and Happy generate Haskell files when run which implement the specified grammars.

6.4 Componests of the Type Checker

6.4.1 Terms

The Haskell representation of terms of the language are given in Listing 6.1. Type lambdas and Lifetime lambdas can be seen as first class citizens here.

```
data Term
    = Var String
    | Lit Int
    | Lam String (Lifetime, Type) Term
    | App Term Term
    | TyLam String Kind Term
    | TyApp Term Type
    | LiLam String Term
    | LiApp Term Term
    | Lt Lifetime
    | Pointer Lifetime Location
    deriving (Eq, Show)
```

Listing 6.1: Haskell representation of Terms.

6.4.2 Kind Checker

The kind checker makes sure that the type expressions in a supplied program are well–formed. The Haskell abstract data type that represents kind expression is show in Listing 6.2.

```
data Kind
    = KnStar
    | KnArr Kind Kind
    deriving (Eq, Show)
```

Listing 6.2: Haskell representation of Kinds.

6.4.3 Lifetime Checking

The syntax of lifetime is show in Listing 6.3.

```
data Lifetime
    = LiVar String
    | LiLit Int
    | LiStatic
    | LiDummy
    deriving (Eq, Show)
```

Listing 6.3: Haskell representation of Lifetimes..

`LiDummy` is used here as a dummy place holder for occurrences of lifetime literals during parsing, as lifetimes are associated with the scope of terms. An initial walk of the tree replaces dummy values with the `LiLit` value that represents the depth of the scope that the lifetime value is found in.

6.4.4 Type Checking

Type checking is rather involved in this this language as the type system almost contains the lambda calculus itself.

```
data Type
    = TyVar String
    | TyInt
    | TyArr Type Type
    | Forall String Kind Type
    | OpLam String Kind Type
    | OpApp Type Type
    deriving (Eq, Show)
```

Listing 6.4: Haskell representation of Types.

6.4.5 Error Reporting

Any part of the program that may result in some kind of error is wrapped in a partial application of Haskell's error monad, `Except`, to a custom error data type:

```
type ThrowsError = Except LangErr

data LangErr
    = ParseError
    | NoMain
    | VarNotFound String
    | WrongKind Kind Kind
    | NotKnArr Kind
    | WrongType Type Type
    | NotTyArr Type
    | NotForall Type
    deriving (Eq, Show)
```

Listing 6.5: Partially applied error monad and language errors.

This has the advantage of being very composeable and also of reducing error handling boilerplate in the program.

6.4.6 Context Management

The lifetime checker, kind checker, and type checker all rely on a variable typing context. These contexts are threaded thought the program using Haskell's environment monad, also known as the Reader monad. The record type holding these contexts is shown in Listing 6.6. Contexts are represented as a map from variable names are strings to some value, using Haskell's own built in strict map data structure.

```
type Ctx = Map.Map String

data Env = Env
    { _typeCtx :: Ctx (Type, Lifetime)
    , _kindCtx :: Ctx Kind
    , _ltCtx :: Ctx Lifetime
    } deriving (Show, Eq)
```

Listing 6.6: Record data type showing contexts.

The monad transformer stack where parsing, lifetime checking, kind checking, and type checking take place is therefore:

```
type Typing = ReaderT Env ThrowsError
```

Listing 6.7: Envirnment and error monad stack.

6.5 Testing

The testing aspect of this project was very important, and was the only way to have some idea that the program worked as expected. Specifically, these tests needed to be automated so that any changes in the behaviour of the program could be checked to make sure that they did not break any previous behaviours. Test programs were manually written, and then loaded from file.

The correctness of the parser, type checker, and evaluator was tested with the HUnit testing framework for Haskell. This tool integrates with the stack build tool so that all tests can be run with a single command. Tests themselves were small programs– these small programs were loaded from file, parsed, and then type checked. The resulting type was then validated against the expected type that was associated with each test file. This system had the advantage that the whole process was checked any time that changes were made to the behaviour of the type checker.

Certain tests were simply to test the parser aspect of the program but others demonstrated key behavior expected from the langauge. These test are covered in more deatil in Chapter 7.

Chapter 7

Key Tests

This section describes key test programs that demonstrate programming with higher kinded types and lifetimes. The syntax the language is described in Chapter 5. Additional constructs have been added for clarity, such as allowing a term to be defined with a name.

7.1 Simple Programs

The programs in this section serve as an introduction to the language syntax and style, without introducing more advanced features. Some of the examples are polymorphic in their type.

7.1.1 Polymorphic Identity Function

This example shows the polymorphic identity function:

```
id: (X, 'a) → (X, 'a)
id = λX:: * . λ'a . λx: (X, 'a) . x
```

Listing 7.1: Identity function

Above, the type of the function is give. All function parameters are associated with a lifetime as well as a type. This is what is meant by the pair `(X, 'a)`. Here `X` is a type parameter which is introduced starting in the scope of the function `λX::* .`. The while `id` term must be applied to type first, as the type lambda expression is outermost. The type that the expression is applied to must be of kind `*`, like `Int` or `X → X`. For example, `id [Int]` is type correct.

The term `λ'a ....`is a lifetime lambda. This term must be applied to some lifetime expression, such as another lifetime variable or a static lifetime.

7.1.2 Multiple Lifetimes and References

More concepts are introduced in this example:

```
add: (&Int, 'a) → (&Int, 'b) → (Int, min('a, 'b))
add = λ'a . λ'b . λx: (&Int, 'a) . λy: (&Int, 'b) . !x + !y
```

Listing 7.2: Addition of two references

Here the dereference operator is used to get the values in the locations held by `x` and `y`. Both of these variables have the type of references of `Int`, demonstrated in the type, `&Int`. These values have an associated lifetime, or the length that they will exist in memory for. The lifetime of the whole resulting expression is the either `'a` or `'b`, whichever exists for the shortest length of time. In other words, whichever lifetime is at the top of the stack.

It is quite useful to have multiple lifetimes bound here. `x` may exist for a different length of time then `y`. A common real world example would be searching for some value in a larger data structure; the value to search for would only

need to exist for the length of the search function, whereas the structure being search might need to exist for a much longer time.

7.1.3 Higher Order Functions

This example shows functions being used as first class values in the language, which is not very interesting in itself, but the type signatures of the functions are of note. In the simply typed lambda calculus, functions may have type `X → X`. Here, functions must annotate the variable that they bind with a lifetime as well as a type. Hence, a function might have the signature `(X, 'a) → (X, 'a)`, meaning that a function takes a value of type `X` that lives for `'a` and return a value of the same type and lifetime.

```
compose: ((B, 'a) → (C, 'a)) → ((A, 'a) → (B, 'a)) → (A, 'a) → (C, 'a)
compose = λA:: * . λB:: * . λC:: * .
     λ'a .
        λf: (B, 'a) → (C, 'a) .
            λg: (A, 'a) → (C, 'a) .
                λx: (A, 'a) . f (g x)
```

Listing 7.3: Function composition

The signatures of `f` and `g` show that they may return values with different types, but the same lifetimes. In fact, all values in this version of `compose` must live for the same length of time.

7.2 Memory safety

The intention of this section is to show how programs written in an unsafe language, like C, can be transformed into a language with memory safety. Specifically, problems in C programs will be highlighted and an equivalent program will be shown that does not compile.

dangle example, calc example that does not typecheck, explain why

7.3 Basic tests

Here basic aspects such as definition

```
double:
double = λX:: * . λa' . λf: (X, 'a) → (X, 'a) . λx:(X, 'a) . f (f x)
```

Listing 7.4: Double Function application

```
quad:
quad = λX:: * . λ'a . double [X → X] a' (double [X] 'a)
```

Listing 7.5: Quadruple application function using double

Chapter 8

Conclusion and Further Work

This section covers ideas that should be implemented in order to make the lanagauge more useful to reason about how these type systems interact.

8.1 Alternatives to Full Higher Kinded Types

Full higher kinded types in Rust may not be necessary to get the benefits commonly associated with this feature. An alternative proposition involves adding type constructors to *traits*, which would be known as *associated type constructors* [3]. Rust already includes an mechanism for associating types with traits, so expanding this feature to include type constructors as well seems like a more natural extension than full higher kinded types, as seen in other languages like Haskell. This section investigates this idea and attempts to define some high level generic constructs using only the proposed idea of Associated Type Constructors.

8.1.1 Traits

Haskell's typeclasses are key in expressing useful high level concepts like Monad, Functor, and Applicative Functors [9]. Traits and typeclasses are a way of providing a generic interface. In Rust, traits programming language fill a very similar role to Haskell's type classes, however traits may only describe concrete types, or types with kind $\star$. If one were to bring higher order polymorphism to a language like Rust then it would be essential to implement type classes, as they are important in leveraging the benefits of higher order polymorphism anyway. There might also be some difficulties in incorporating type classes into a language with higher order polymorphism and Rust's model of memory management.

In essence, generic container objects must keep track of the lifetimes of the objects that they contain. In order to understand some of the problems with implementing generic interfaces for containers (such as Functor, from Chapter 3) some additional concepts are needed. Examples are given in Rust.

8.1.2 Associated Types

Associated types refer to types that may be defined in a trait, and used by methods that are defined by that trait. This example from the Rust documentation [5] illustrates this concept.

Suppose that we wanted a trait to describe a graph, with generic node and edge type:

```
trait Graph<N, E> {
    // ... useful graph methods defined here
    fn edges(&self, &N) -> Vec<E>;
}
```

The graph has been parametrized with the types `N` for the type of nodes and `E` for the type of edges. While this works well for operating over a generic graph it is a bit cumbersome. For example, if we wanted to write functions in Rust that operate over `Graph` we would have to write:

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 {
    // ... distance implementation
}
```

There are quite a few type parameters in this function signature and it impedes understanding. In particular, it is not clear that `N` and `E` are associated with `Graph`. `distance` does need to reference these types in its signature, though. To solve this issue Rust introduces associated types. These are illustrated in a this revised `Graph` example:

```
trait Graph {
    type N;     // type of nodes
    type E;     // type of edges

    fn edges(&self, &Self::N) -> Vec<Self::E>;  // Self references this trait
}
```

Note that the trait now has type definitions included in its definition. Graph no longer takes type parameters, but does still reference the generic types `N` and `E`. The signature of the `distance` function that operates over this new graph looks like:

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> u32 {
    // distance implementation
}
```

This function signature, as well as being more brief, also makes it very clear that the node and edges types are associated with the graph type.

Associated types define types that are part of an interface. Note that in the current version of Rust, associated types may only be of kind $\star$, i.e. they may not be type constructors. Expanding associated types to include type constructors may have some interesting benefits, commonly associated with full higher kinded types [4].

8.1.3 Iterators and Generic Collections

There are several motivations for including associated type constructors.

Iterators

Iterators allow programmers to traverse containers in a safe, generic manner. They are an alternative accessing an collection with indexing operations and are more suited to traversing collections that do not provide effective random access, like lists or trees. Many types of collection may provide iterators and so providing an iterator interface makes code more generic by not associating operations on collections with the implementation of those operations.

In Rust, a trait that describes what operations an iterator should provide can be written like:

```
trait Iterator {
    type Item;  // Associated type item describes what type the iterator will return
    fn next(&mut self) -> Option<Self::Item>;
}
```

The method defined here, `next`, says that any class that implements this trait must define how collection elements are accessed. Elements of type `Item` that are held by a collection are returned when there are more elements to access.

Attempting to Define Generic Collections

Making `Iterator` a trait would allow code to be written that can separate the logic of using elements of a collection from the implementation of how those elements are accessed. To expand on this idea, it would be useful to have an interface for all types of collections that provide a few key operations. If this could be done, it would further separate implementation details from logic of using collections. This would facilitate code reuse and allow code to be swapped out or replaced more easily because the collection we wanted to use would only have to conform to a given interface, or trait. One of the key methods that the trait would provide would be some way of accessing elements of the collection in a generic way, probably using iterators. It might look something like:

```
trait Collection<T> {
    // Create an empty collection of type T
    fn empty() -> Self;

    // Add an element of type T to this collection. For linked lists, it might
    // prepend to the collection, for sets it might insert, etc
    fn add(&mut self, value: T);

    type Iter: Iterator<T>;     // An iterator for this collection would also
                                // be of type T, becuase it would reference
                                // a single element of this collection at one
                                // time

    // Get a reference to an iterator of this collection so you can do things
    // with the elements of this collection by calling the next method defined
    // in the Iterator interface
    fn iterate(&self) -> Self::Iter;
}
```

Now we can try and define a collection that implements all of the operations described in this trait. Here's what a minimalistic linked list might look like in Rust:

```
pub struct List<T> {
    cell: Option<Box<ListCell<T>>
}

struct ListCell<T> {
    value: T,
    next: List<T>
}
```

Some methods can be implemented for this collection:

```
impl<T> List<T> {
    pub fn new() -> List<T> {
        List { cell: None }
    }

    pub fn prepend(&mut self, value: T) {
        // Prepend an element to this collection...
    }

    pub fn iter<'iter>(&'iter self) -> ListIter<'iter, T> {
        ListIter { cursor: self }
    }
}

// Define a list iterator that does what an iterator should, specifically for
// lists.
pub struct ListIter<'iter, T> {
    cursor: &'iter List<T>
}

impl<'iter, T> Iterator for ListIter<'iter, T> {
    // Here the Item associated type is instantiated, from the Iterator trait
    // Note that the item is a reference to an element T, that lives for the
    // lifetime 'iter
    type Item = &'iter T;
    fn next(&mut self) -> Option<&'iter T> {
            // return a Option with type T, which is a reference to an element
            // in the collection, or the same type as Item
        }
    }
}
```

This implementation of an iterator for a list returns a reference to an `Item` from the list. `Item` needs to have a lifetime because any method that uses an element from this collection needs to know how long the collection lives for. A reference to an element in the collection could not be used after the collection itself has been deallocated. It does not matter what the type of that element is though, only that it lives long enough. `ListIter` takes a lifetime parameter accordingly.

`List` should implement `Collection`, because it obeys all the rules for this trait. Other types, such as trees or different types of list could also implement `Collection`. Here we try and define the implementation of `Collection` for `List`:

```
impl<T> Collection<T> for List<T> {
    fn empty() -> List<T> {
        List::new()
    }

    fn add(&mut self, value: T) {
        self.prepend(value);
    }

    fn iterate<'iter>(&'iter self) -> ListIter<'iter, T> {
        self.iter()
    }

    // This will not compile because there is no reference to the lifetime
    // of the type in the generic collection. The Collection interface does
    // not provide a way of accessing the lifetime of a object associated with
    // a collection that lives for a certain length of time.
    type Iter = ListIter<'iter, T>;
}
```

There is a large problem in this implementation. A collection contains elements of type `T`, and provides access to an iterator that should allow access to those elements. In the case of `List`, `Iterator` returns a reference to the element which lives for the same length of time as the whole collection. This is problematic though because there is no way to reference the lifetime of a collection in the trait above.

This shows how modelling a generic collections interface can be difficult. In the case of `Iterator`, the lifetime of the element could be passed in to the instance of the trait. This cannot be done with the implementation of `Collection`. Specifically, `List<T>` does not provide enough information to instantiate `ListIter<'iter, T>`.

Associated Type Constructors

To solve this problem, the type of the item that the iterator returns needs to be associated with a lifetime. Being able to include type constructors in classes as well as concrete types would allow this. For example:

```
trait Collection<T> {
    fn empty() -> Self;
    fn add(&mut self, value: T);

    // The type of Iter is now parametrized with the lifetime of the element
    // that it contains
    type Iter<'iter>: Iterator<T>;
    fn iterate<'iter>(&'iter self) -> Self::Iter<'iter>;
}
```

Now `Iter` is a type constructor rather than a concrete type, and must be instantiated with a concrete type before it is used. As a brief aside, lifetime variables are actually another *kind* in Rust:

Kinds

K	::=	$*$	kind of concrete types
	|	$lifetime$	kind of lifetimes

Figure 8.1: Kinds in Rust

This can be seen when passing polymorphic type arguments to functions in Rust. For example:

```
fn foo<'a, T>(x: &'a T) -> u32 {
    // do stuff
}
```

This function takes a polymorphic lifetime argument and a type arguments. `Iter<'iter>` is a type constructor, or function at the level of types, and hence has kind `lifetime -> *`.

With associated type constructors, the collection implementation for `List` could look like:

```
impl<T> Collection<T> for List<T> {
    // same methods as before

    type Iter<'iter> = ListIter<'iter, T>;
    // Use of iter now must be parameterized by a lifetime, which can be used
    // by ListIter. Information can now be generically passed through.
}
```

This solves the problem of how to get the lifetime information to `ListIter`. Methods can now be written that are generic over collections:

```
fn add_one<C>(xs: &C) -> C
    where C: Collection<u32>
{
    let mut xs_plus_ones = C::empty();  // Collection trait defines empty
    for &x in xs.iterate() {            // xs is a collection, which defines iterate
        xs_plus_ones.add(f + 1);
    }
    xs_plus_ones
}
```

This function could operate over trees, lists, sets, or anything else that operates over a collection. This is very useful, however there are still problems with this approach. In Chapter 3, we tried to implement `Functor`, or an interface for a container that only provided one method, `map`. The issues arise when we try and implement a generic map function `Collection`, similarly to how map was implemented for `Functor` in Chapter 3.

```
fn map<A, B>(f: &Fn(A) -> B, xs: &Collection<A>) -> Collection<B> {
    let mut out = Collection<B>::empty();
    for &x in xs.iterate() {
        out.add(f(x));
    }
    out
}
```

The problem here is that this function can return *any* type that implements collection, not the same type of collection that we called `map` on. This means that there is very little that can be done with the result of this function, without casting or explicit annotation. This is especially problematic if calls to `map` and other higher order functions like `filter` are chained, which is a common idiom.

Referencing Chapter 3, what we really want to be able to do have a signature like:

```
// Note that C is defined in the type parameters for this function
fn map<A, B, C: Collection>(f: &Fn(A) -> B, xs: &C<A>) -> C<B> {
    // same as before
}
```

In effect we want to say that the type parameter `C` should itself be a type constructor. This constructor is referenced in the rest of the signature and the parameter `xs` and the return type now have to be the same type of collection. However this problem can be solved without using full-blown higher kinded types, and instead using the associated type constructors defined earlier in the chapter.

Type Families

Using associated type constructors and a few extra traits, the relationship between the type constructor a list of parameters and the return type of a function can be defined. The extra traits needed are described below:

```
trait CollectionFamily {
    // CollectionFamily defines one associated type constructor
    type Member<T>: Collection<T>;
}

struct ListFamily;

impl CollectionFamily for ListFamily {
    // instantiate list families with with the type of lists holding generic
    // elements
    type Member<T> = List<T>;
}
```

Family refers to the collection type without referring to its elements, e.g. any type of `List` might be a collection family. This can be used to implement a proper map function:

```
fn map_family<A, B, F: CollectionFamily>(xs: &F::Collection<A>) -> F::Collection<B> {
    let mut out = F::Member::empty();
    for &x in xs.iterate() {
        out.add(f(x));
    }
    out
}
```

This function can then be called with a family of types, like `ListFamily`, as a type parameter. The function would then take a `List<A>` as an argument and return a `List<B>`.

This is one of the main motivations for including higher kinded types in a language. However, lots of the benefits of higher kinded types could potentially be modelled by using associated type constructors. The would solve some problems show in Section **??** related to lifetimes, which would not clearly be solved by higher kinded types alone. This system potentially has the other benefits. Higher kinded traits, a la Haskell's type constructor classes, would not need to be implemented. These could potentially confuse newcomers to the language, as well as possibly making type inference more difficult [4].

A future project would be to implement a language with lifetimes and type inference, and incorporate Rust-style traits. There would be no higher kinded types, in the style of Haskell, would not be built into the language. Traits could only be written over concrete types. Associated type constructors could then be added and the propositions in this section could be tested.

8.2 Extensions to This Project

Some other extensions to the lambda calculus used in this project should be modelled in order to better reason about the influence higher kinded types would have on Rust.

8.2.1 Lifetime Elision

Lifetime elision means that lifetimes can be omitted in cases where they are obvious from their context of use. Type signatures often do not need to be annotated with types because default options can be assumed that agree with what the programmer is trying to express. It is a separate concept from inference. Elision significantly cuts down the

amount a programmer has to write and reduces unneeded information. Manual lifetime annotations such as found in this project are not very practical for general programming.

By default in Rust, every value in a function's parameters that has a separate lifetime. If a function only has a single input lifetime, then that lifetime is used for every output lifetime.

8.2.2 Type Inference

Type inference refers to automatically deducing types of expression rather than making the programmer explicitly annotate them. It is common in statically typed functional languages. It makes writing programs easier to write and read, as less space used writing down types of expressions that are obvious or can easily be inferred from context.

However, global type inference for System F has been proved to be undecidable [12] [11]. In practice, languages like Haskell that do have global type inference rely on *let-polymorphism*, which has proven to be the sweet spot between expressive power in a ease of use as a programming language. Lots of languages with type inference, including Rust, go down this route. If one were to design a language that included lifetime annotations and higher kinded types, type inference would probably be a good idea as the types of expressions and function signatures could become very large and unwieldy. Therefore it would make sense to include let-polymorphism.

8.2.3 Record Types

Record types, called structs in Rust, are a way of combining existing types to create more complex type. These more complex types can then be treated as a single datatype. They are commonly used to store related data and treat that data as a single entity.

Bibliography

[1] *Alex: A lexical analyser generator for Haskell*. https://www.haskell.org/alex/. Accessed: 2016-10-18.

[2] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997. ISBN: 0-521-58775-1.

[3] *Associated type constructors (a form of higher-kinded polymorphism)*. https://github.com/rust-lang/rfcs/pull/1598/commits/406400c082c4989e94b04a423988b13ac12d1afa. Accessed: 2017-4-22.

[4] *Associated type constructors, part 1: basic concepts and introduction*. http://smallcultfollowing.com/babysteps/blog/2016/11/02/associated-type-constructors-part-1-basic-concepts-and-introduction/. Accessed: 2017-4-22.

[5] *Associated Types*. https://doc.rust-lang.org/book/associated-types.html. Accessed: 2017-2-20.

[6] *Happy: The Parser Generator for Haskell*. https://www.haskell.org/happy/. Accessed: 2016-10-18.

[7] *Haskell: An advanced, purely functional programming language*. https://www.haskell.org/. Accessed: 2016-10-18.

[8] Matthew Hertz and Emery D Berger. "Quantifying the performance of garbage collection vs. explicit memory management". In: *ACM SIGPLAN Notices*. Vol. 40. 10. ACM. 2005, pp. 313–326.

[9] Mark P Jones. "A system of constructor classes: overloading and implicit higher-order polymorphism". In: *Journal of functional programming* 5.1 (1995), pp. 1–35.

[10] Amit Levy et al. "Ownership is theft: experiences building an embedded OS in rust". In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. ACM. 2015, pp. 21–26.

[11] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004. ISBN: 0262162288.

[12] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[13] *RAII*. http://en.cppreference.com/w/cpp/language/raii. Accessed: 2016-10-18.

[14] *The Haskell Tool Stack*. https://docs.haskellstack.org/en/stable/README/. Accessed: 2016-10-18.

[15] *The Rust Programming Langauge*. https://www.rust-lang.org/en-US/. Accessed: 2016-10-18.

[16] Jesse A. Tov and Riccardo Pucella. "Practical Affine Types". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 447–458. ISSN: 0362-1340. DOI: 10.1145/1925844.1926436. URL: http://doi.acm.org/10.1145/1925844.1926436.