Programming Language comparison

University of Brighton
Richard Goodman
13842540

January 12, 2017

Abstract

In this report we compare and contrast two programming languages: Java and Haskell. We will explore different programming paradigms to highlight similarities as well as differences. Finally, a reflection will be composed on the recorded findings.

1. Introduction

Throughout the time-line of computing technology there has always been a primary need to write functions in a programming language. The realm of programming languages has expanded since, resulting in not only different languages, but also various programming paradigms.

2. Overview

The two languages chosen for this report consist of an object-orientated programming language and a functional programming language.

2.1. Java

Starting with Java, it is an object-orientated language released in 1995 by Sun Microsystems *(now Oracle)*. Arguably one of the most known and used languages out there. Some argue that it is a weakly-typed language due to the ability to cast, however all variables need to have a type declared, thus it being a strongly typed language.

2.2. Haskell

First created in 1990, however, a stable release announced in 2010. Haskell is a functional language and provides functionality that is being recognised and influencing various differing languages to adopt these properties. It is also a strongly typed language and features lazy evaluation.

3. Inheritance

It's well known, with Java being an Object Orientated Language *(OOL)*, there is the classic saying 'Everything is an object'. With an OOL a form of inheritance is introduced. There are languages out there that offer multiple inheritance, Java however, is not one of them.

This is to avoid the multiple inheritance problem, or as it is commonly known as the diamond problem. The diamond problem is when a child class inherits from two parent classes, which themselves inherit from a parent class, as seen diagrammatically in figure 1. If attributes from the parent classes have the same name the compiler will throw back an error due to it not knowing which attribute to use.

As for Java, "Java supports only a single inheritance class hierarchy, but allows multiple inheritance of interfaces" [1]. Interfaces are not to be considered as objects themselves, but rather a collection of abstract methods. One advantage of an interface is allowing certain classes implementing methods that not necessarily all objects will implement. An example of this can be seen in Appendix A.

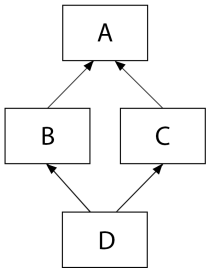


Figure 1: *The diamond problem*

When looking at inheritance with Haskell its a completely different perspective. From an email found on the Haskell's website [2], Paul states that in the given example *(Found in Appendix B)*, is not the same as classes as commonly found in OO languages. Instead 'Man' and 'Women' are data constructor functions which return the type Person, there is no form of inheritance here despite looking at the code found in Appendix B hinting a form of inheritance.

However, inheritance can be achieved in Haskell. Lipovaĉa tells us that by creating a type class it acts as "an interface that defines some behavior. If a type is an instance of a type class, then it supports and implements the behavior the type class describes." [3]. By creating type classes in conjunction with data types, one can achieve inheritance. An example can be found in Appendix C, where by using the Equalizer type class and creating a data type of 'PrimaryColor', we can create an instance of Eq with a type variable being passed through *(In this case, PrimaryColor)*. With this we can inherit the behaviour from the Eq type class and append it to our data type.

4. Polymorphism

A hint of Polymorphism has already been shown in the inheritance code examples, but without any detailed explanation on what it is, nor how the languages deal with it.

Polymorphism is a code construct that has become highly popular in programming languages in the last decade. Polymorphism is when "Type systems that allow a single piece of code to be used with multiple types" [4], with this said, there are different variations of polymorphism, the two most popular being:

- Parametric polymorphism - *A piece of code to be typed "generically," using variables in place of actual types.*
- Ad-hoc polymorphism - *Allows a polymorphic value to exhibit different behaviors when "viewed" at different types.* [1]

The languages discussed in this paper make use of polymorphism. Haskell in particular has the capability to exploit both parametric and Ad-hoc polymorphism, whereas Java can only make use of parametric polymorphism.

By utilising the benefits of polymorphism, it allows the programmer to write reusable code. A simple example of polymorphism can be shown in Haskell:

```
Prelude> :t head
head :: [a] -> a
```

Here **'a'** is a generic type, where it doesn't matter what type the variable is, it will be validated by the compiler and executed.

Of course this simple example only paints a small picture for the advantages of parametric polymorphism. In Java polymorphism is commonly known as 'Generics', and an example can be found in Appendix D. Notice how in the example, when adding items to one list, the items themselves have various types. A similar effect can be achieved by storing objects, for example:

```
ArrayList<Animal> al = new ArrayList<
    Animal>();
```

Here we are creating an ArrayList of 'Animal' objects, by using inheritance and creating child classes that extend Animal, we can store these children objects inside this ArrayList.

Now looking at Ad-hoc polymorphism, what is it? And how does it compare to the standard parametric polymorphism?

[1] Both descriptions can be found in citation [4], *page 340*.

4.1. Ad-hoc polymorphism

Also known as overloading, it introduces an extra level of complexity for the compiler. Augustsson explains how overloading in Haskell produces the outcome of higher-order functions, suggesting that they are "hard to handle efficiently because much less is known about them at compiler time" [5]. *(Also discussed on the Haskell wiki [6])*.

Despite the performance disadvantages, Ad-hoc polymorphism provides some powerful advantages. Serrano Mena introduces Ad-hoc polymorphism by providing the type declaration of a Map function.

```
Prelude> :t M.insert
M.insert :: Ord k => k -> a -> M.Map
     k a -> M.Map k a
```

He states "The purpose of **Ord k** is to constraint the set of possible types that the k type variable can take." [7]. In this example, a function is passed through as a parameter *(as previously mentioned as higher-order functions)*. Looking at the example provided, the Ord type class is telling the compiler that the type must provide implementations of comparison operators. Looking at this, another interpretation can be displayed by the following:

```
foo :: [t] -> t -> Bool
```

This tells us that foo will not work for all types t, but for only types t which have the ability to support equality operations to them.

Another feature of Ad-hoc polymorphism as previously mentioned is the ability to provide different behaviours for different types. Doing so can not only give the ability to write different methods for any chosen specific type, but also give the ability to write different return statements. Although you can argue this doesn't necessarily create code reuse, if you as the programmer create a generic polymorphic function, which in turn you extend by create specific alternatives for certain types, does that not feature code reuse?

5. Memory Management

Memory management is a crucial element for programmers that must be taken into consideration when writing a program. In particular Java's main method of memory management is the use of Garbage Collection *(GC)* that is automatic in any Java application.

As for Haskell, due to it's lazy evaluation, it means "that expressions are not evaluated when they are bound, not where they are in your source code" [8]. Sewell also talks about how this is beneficial to the programmer, as the

as the whereabouts of the execution of the code takes place isn't a main priority. However, he also explains how it is a draw back as it increases the risk of memory leakage.

5.1. Memory Leakage

Memory leakage is a prone problem. The definition of a memory leak is when a program has trouble getting rid of deallocated memory. As a whole this can affect the performance of the program as well as using more memory than intended. What makes it more challenging is that it isn't always noticeable to the programmer.

With Java being reliant on GC, one can assume that memory leakage wouldn't occur. Salnikov-Tarnovski informs us, when creating a new object, the memory allocation is handled by the JVM *(Java Virtual Machine)* [9]. Despite this, if the GC itself doesn't know whether or not an object is no longer in use and fails to remove it from memory then a memory leakage will be present. An example of a memory leak in Java can be found in appendix E.

As previously mentioned, with Haskell being a lazy evaluated language, it can arise memory leaks fairly easily. Likewise with Java, Haskell has the ability to use garbage collectors *(although not compulsory)*, thus reducing the chances of memory leakage. An easy example of a memory leak in Haskell is as follows:

```
foldl (+) 0 [1..n::Integer]
```

What seems a simple and faultless piece of code, provides a classic example of memory leakage. Instead of only returning the final result of x, it stores the sum calculated as the compiler does not know whether the sum(s) would be needed at a later date;

$$x = (1 + ... + n)$$

To avoid leakage *(in this particular case)*, forcing a strictness evaluator to the code will force the result to execute the calculation of the sum(s).

```
foldl' (+) 0 [1..n::Integer]
```

One final point to make with Haskell is the ability to test a program to see any potential leakage. One can create a test file in which an overall memory allocation is known. Executing the test file with extra parameters, *(Where M2m translates to 2Mb and can be changed to any number)* can force a size restriction on the heap to see any memory leaks.

```
\$ ./testFile +RTS -M2m -RTS
```

6. Garbage Collection

In the previous section the word Garbage Collection was mentioned a few times. Garbage collection is a popular system used in programming languages using heuristics to handle memory deallocation as briefly explained.

Java uses a generational GC by default. If objects become eligible for removal, the GC will deal with it. To make an object eligible, one must remove any live references to the object leaving the object on the heap. Sierra and Bates tell us that there are 3 ways to get rid of an objects reference [10]:

1. The reference goes out of scope, permanently.

```
void go() {
 Life z = new Life();
}
```

2. The reference is assigned another object.

```
Life z = new Life();
z = new Life();
```

3. The reference is explicitly set to null.

```
Life z = new Life();
z = null;
```

As stated above, the reference on the heap must be removed in order for the object to be dealt with the GC. This can be visually represented with a code example in Appendix F.

The benefit of having a GC is it minimizes the risk of memory leakage, making it a more profound choice of language for the programmer, as it is one less obstacle for them to take into consideration fully *(Obviously as we have discussed, they will have to take it into consideration, but the chances are significantly decreased)*.

As for Haskell, GC is an interesting subject, depending on the compiler you are using will present a different GC. One of the most common Haskell compilers, GHC, has had "At least three different garbage collection schemes" [11] [2]. It's interesting to note is how these each GC implemented presented different approaches; *(a one-space in-place compacting collector, a two-space stop and copy collector and a generational garbage collector)*. However, due to certain limitations of the GHC compiler versions, in particular 3.0x to 4.0x meant that a While and Field GC was unable to be implemented. [11].

When looking at the Haskell wiki on GC, a paper is referenced presenting the latest approach to the chosen GC applicable for the current GHC version. Marlow presents a parallel generational-copying GC with a block-structured heap. "By using a block-structured memory allocator, it provides a natural granularity for diving the work of GC between many threads" [12].

7. Concurrency

Concurrency is a powerful element of programming that has the ability to compute multiple

[2] At the time of this paper being published.

tasks at the same time. With the development of processors leading to multiple cores on a singe chip, it's becoming a norm for applications to follow this. Both languages compared in this short paper can make use of concurrency.

Concurrency can become messy, however. If a program is using multiple threads and sharing variables, the results can vary due to threads being out of sync with one another.

This is exactly what Winterberg discusses [13]. Java supports the use of thread synchronization *(parallelism)*, this removes the risk of threads running 'out of loop' between one another, following his point an example of this can be seen in appendix G.

Concurrency is also able in Haskell, and makes use of parallelism also. however, due to it's lazy evaulation, more precautions need to take place. This is because evaluations aren't given till they are absolutely most needed, as described above with the foldl example.

Marlow introduces the Eval Monad, showing 3 seperate examples how the computation differs found in appendix H [14].

As seen, in the appendix, depending on which method will result in different return positions. Marlow states that "rpar/rseq is unlikely to be useful as the programmer doesnt know in advance wihch computation will take the longest." [14], and that 'rpar/rpar' or 'rpar/rseq/rseq' are more preferable, depending on the criteria of the program.

8. Conclusion

This journey has provided a new insight to Haskell. Prior to this I had no knowledge of the language and has been interesting to learn about its style and how the language works in comparison to Java.

Apart from the obvious difference Haskell has, such as writing programs in less lines. What I found is that the two languages share a lot of similarities, but interpret them in their own way.

Personally, I wasn't aware Ad-hoc polymorphism was a form of polymorphism, let alone Java not being able to implement it. From my findings, I can see the benefits of Ad-hoc polymorphism and the potential it can bring to the programmer.

Overall I have found with Haskell being a much higher-level language than Java, it provides a more abstract approach to programming. However, I found it personally difficult to learn and understand it as a language as it requires a more profound programming knowledge to learn.

Nevertheless, I have a new respect for the language and want to experiment further. I do find it interesting how the language isn't as approached than Java, nor a big topic of discussion due to the benefits it provides.

But with it begin considered 'new', I can only hope its pressed more to be used in the computing industry.

References

[1] Niemeyer, P., Knudsen, J. and Posner, J. (2000) Learning java: [covers java 1.3]. Edited by Mike Loukides.
New York, NY, United States: O'Reilly Media, Inc, USA.

[2] Paul Sargent,
[Haskell-beginners] Equivalent of inheritance in Haskell.
`https://mail.haskell.org/pipermail/beginners/2010-December/006084.html`
Accessed 19/12/2016.

[3] Lipovaĉa, M, (2011),
Learn you a Haskell for great good!: A guide for beginners.
San Francisco, CA: No Starch Press San Francisco, CA.

[4] Pierce, B.C. (2002),
Types and programming languages. Cambridge, MA: The MIT Press.

[5] Lennart Augustsson,
Implementing Haskell overloading. *Date*

published unknown
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.54.7639&rep=rep1&type=pdf
Accessed 28/12/2016.

[6] Author unknown,
Performance Overloading. Last modified 30/12/2010.
https://wiki.haskell.org/Performance/Overloading

[7] Mena, A.S. (2014).
Beginning Haskell: A project-based approach.
Berkeley: Apress.

[8] Will Sewell, 29/01/2016.
Memory profiling in Haskell.
https://blog.pusher.com/memory-profiling-in-haskell/
Accessed 04/01/2017

[9] Salnikov-Tarnovski, N., 15/02/2012.
What is a memory leak?
https://plumbr.eu/blog/memory-leaks/what-is-a-memory-leak
Accessed: 05/01/2017

[10] Sierra, K. and Bates, B. (2005).
Head first java. 2nd edn.
Boston, MA, United States: O'Reilly Media, Inc, USA.

[11] Cheadle, A, 18/06/1999.
Incremental Garbage Collection for Haskell
An MEng Computing Final Year Project Report
Department of Computing, Imperial College, London.

[12] Winterberg, B., 30/04/2015.
Java 8 Concurrency Tutorial: Synchronization and Locks.
http://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/
Accessed: 08/01/2017.

[13] Marlow, S., Harris, T., James, R., Peyton Jones, S., (2008).
Parallel Generational-Copying Garbage Collection with a Block-Structured Heap.
http://simonmar.github.io/bib/parallel-gc-08_abstract.html
Accessed 06/01/2017.

[14] Marlow, S. (2013).
Parallel and concurrent programming in Haskell.
Sudbury, MA, United States: O'Reilly Media, Inc, USA

Appendix A

An example of a short Java application using inheritance and implementing an interface.

```java
//Interface.java
public interface Pet {

  public abstract void bePraised();

  public abstract void beScolded();

}
```

```java
//Feline.java
public class Feline {

  public void meow() { System.out.println("Meow"); }

}
```

```java
//Cat.java
public class Cat extends Feline implements Pet {

  @Override
  public void bePraised{ System.out.println( "Your cat as been
     praised." ); }

  @Override
  public void beScolded{ System.out.println( "Your cat has been
     told off." ); }

}
```

Appendix B

Example taken from [2], discussing inheritance in Haskell. *(Example was slightly adjusted to be less offensive as stated by Richard Mittel, https://mail.haskell.org/pipermail/beginners/2010-December/006085.html (Accessed 19/12/2016)).*

```haskell
data Person = Man {name :: String, age :: Int} | Woman {name :: String, age :: Int}

  meet :: Person -> Person -> String
  meet (Man m _) (Women w _) = m ++ ", " ++ w ++ "have met."
```

Appendix C

An example of inheritance in Haskell *(The class Eq is already implemented in the default Haskell library, but is shown for the purpose of this example)*. This example can also be found in citation [3] *(Page 139, following the TrafficLight example)*.

```
--Eq Class
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
--inheritance.hs
data PrimaryColor = Cyan | Magenta | Yellow
  deriving Show

instance Eq PrimaryColor where
  Cyan == Cyan = True
  Magenta == Magenta = True
  Yellow == Yellow = True
  _ == _ = False
```

Appendix D

An example of parametric polymorphism in Java. This short program creates a List of various Items which can have any type, and will return the list and print it to the console.

```
//Main.java
import java.util.*;
public class Main {

    public static void main(String[] args) {

        List l = new List();

        l.add("Foo bar");
        l.add(false);
        l.add(10);

    //This is only here to show a new ArrayList being created, and
       appending all the items from the original list.
        ArrayList<Item> test = new ArrayList<Item>();
        test = l.getList();

        for ( Item i : test ) { System.out.println("Item: " +
           i.getData() + "." ); }

    }
}
```

```java
//List.java
import java.util.*;

public class List<T> {

    ArrayList<Item> list = new ArrayList<Item>();

    public void add(T data) {
        Item<T> item = new Item<T>(data);
        list.add(item);
    }

    public ArrayList<Item> getList() { return list; }

}
```

```java
//Item.java
public class Item<T> {

    private T data;

    public Item(T data) {
        this.data = data;
    }

    public T getData() { return this.data; }

}
```

Output

```
Item: Foo bar.
Item: false.
Item: 10.
```

Appendix E

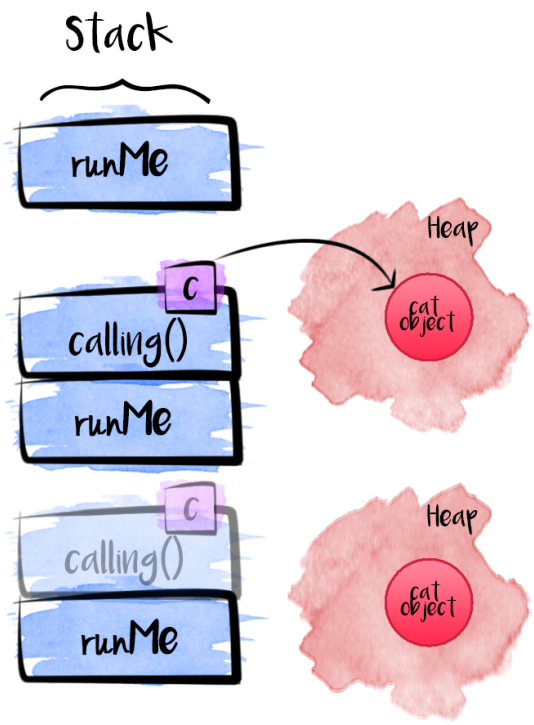
An example showing a memory leak in Java, if you see there is a HashMap storing the original inputs, however, as they are never used the object is never de-referenced, causing a memory leak. Although the HashMap is an useless variable in this example, it is used to provide a simple example.

```
1  import java.util.Scanner;
2
3  public class Palindrome {
4    private Map memory = new HashMap();
5
6    public boolean checkString(String s) {
7      String reverse = new StringBuilder(s).reverse().toString();
8      memory.put(map.size(), s);
9      return (s.equals(reverse)) ? true : false;
10   }
11
12   public static void main(String[] args) {
13
14     Palindrome p = new Palindrome();
15     while (true) {
16       Scanner sc = new Scanner(System.in);
17       System.out.println("Enter a string to see if it's a
             palindrome");
18       String input = sc.nextLine();
19       if (p.checkString(input) == true) ? System.out.println(input
             + " is a palindrome.") : System.out.println(input + " is
             not a palindrome.");
20     }
21   }
22 }
```

Appendix F

A short code example along with a visual representation of how Garbage collection in Java is activated.

```
1 public class GC {
2   public void runMe() {
3     calling();
4   }
5
6   public void calling() {
7     Cat c = new Cat();
8   }
9 }
```



By looking at the code, when runMe is executed, it is pushed onto the stack. Following this, the method calling is called which again will be pushed onto the stack. Here it declares a reference to the variable c, which creates a new object of Cat onto the Heap.

Finally, when the calling method is finished, it pops off the stack, making the reference c dead. As the cat object on the heap is now abandoned, the Garbage collector deals with this to remove it from memory.

Appendix G

An example of synchronised threads in Java, idea taken from [13].

```
import java.util.concurrent.*;
import java.util.stream.*;

public class Main {

  private static int count;

  public static void main(String[] args) {
    syncTest();
    nonSyncTest();
  }

  private static void syncTest() {
    count = 0;

    ExecutorService exec = Executors.newFixedThreadPool(2);

    IntStream.range(0, 10000)
            .forEach(i -> exec.submit(Main::incrementalSync));

    exec.shutdownNow();
    System.out.println("Synchronised: " + count);
  }

  private static void nonSyncTest() {
    count = 0;

    ExecutorService exec = Executors.newFixedThreadPool(2);

    IntStream.range(0, 10000)
            .forEach(i -> exec.submit(Main::incremental));

    exec.shutdownNow();
    System.out.println("Non Synchronised: " + count);
  }

  private static synchronized void incrementalSync() { count ++; }

  private static void incremental() { count ++; }

}
```

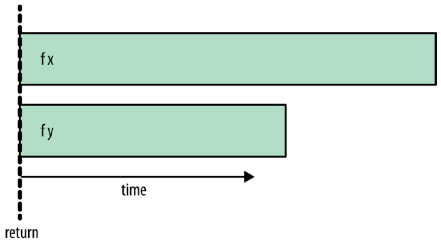
Output

```
Synchronsed: 10000
Non Synchronised: 9984
```

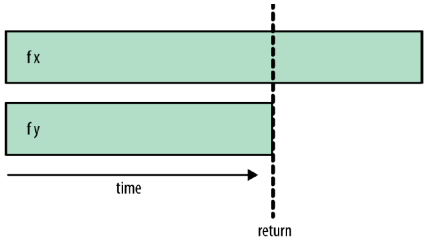
Appendix H

Example of the Eval Monad and it's different computation results.

```haskell
--rpar/rpar
runEval $ do
  a <- rpar (f x)
  b <- rpar (f y)
  return (a,b)
```




```haskell
--rpar/rseq
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  return (a, b)
```




```haskell
--rpar/rseq/rseq
runEval $ do
  a <- rpar (f x)
  b <- rseq (f y)
  rseq a
  return (a,b)
```