Comparing Code extraction from Agda to Java to existing Methods

Lukas Zimmerhackl
Supervisor(s): Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 18, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Comparing Code extraction from Agda to Java to existing Methods

LUKAS ZIMMERHACKL*, Delft University of Technology, The Netherlands

Dependent programming languages such as Agda show a lot of promise in creating new ways of writing code, but currently suffer from a lack of support and features. In this paper we attempt to create a new back-end for Agda targeting Java which has a huge and thriving ecosystem.

We implement the new back-end for Agda in Haskell and we describe the benefits and drawbacks of targeting Java. Firstly we go into the existing methods of compiler Agda, then we go into how to compile Agda to Java and what the main challenges where creating the compiler and what solutions were implemented to solve these. Afterwards Agda2Java is compared to the existing methods of compiling Agda code by means of benchmarks and analyzing the execution time. We show that at its current state, Java does not seem to be a promising back-end for Agda, but that there is work being done on Java that might change this perception.

Additional Key Words and Phrases: Code extraction, Compilation, Agda, Java, Agda2Java

1 INTRODUCTION

While writing code, you want to focus on the problem you are trying to solve, as much as possible. That is why we use all sorts of tools to help us with coding, such as code highlighting, code auto-completion and especially error checking. Error checking is especially useful when it points out the errors to you immediately whenever you write a line of code. These types of errors are called compile-time errors and include syntax errors, lexical errors and type errors.

Type errors occur due to the nature of many programming languages such as Java, which use a static type system. This allows the Java compiler to catch errors while the programmer is writing code. The Java type system ensures that functions are always called with the right argument type, variables are assigned to the correct type and many other type-related problems [Drossopoulou and Eisenbach, 1997].

However, the Java type-system is not all-powerful and does not catch all errors. Errors such as getting the first element of an empty list, are not handled by the Java type-checker. There is however a variety of type systems that do catch these errors, dependent type systems. Dependent type systems allow types to depend on a term of another type. This idea will be further explored insection 2, but for our example it means that dependent types allow us to write functions that only accept non-empty lists.

One implementation of a dependent type system can be found in Agda, a dependently typed language. Once an Agda program has been type-checked, we can be sure no function will ever receive an empty list, when a non-empty list is expected. Agda currently has no way of directly compiling to byte code but instead compiles to another programming language, which then compiles to bytecode. This process is called code extraction, but we will refer to it in this paper as compiling. For the latest version of Agda, there are two code extractors, the MAlonzo GHC back-end and the JavaScript back-end [Team, 2021]. The MAlonzo back-end is currently the norm when it comes to Agda development, whereas JavaScript still has many limitations. A new back-end that compiles to Java code could help Agda's development immensely, this would allow Agda to benefit from the potential speedup Java could bring, as well as allowing Agda code to interact with the entire Java ecosystem.

Therefore my research question is: *"Under which conditions is Java a suitable target language for code extraction from Agda compared to existing methods?"*. Firstly, to answer this question, existing

Author's address: Lukas Zimmerhackl, Delft University of Technology, Van Mourik Broekmanweg 6, 2628XE, Delft, The Netherlands, L.K.Zimmerhackl@student.tudelft.nl.

methods of compiling Agda will be looked at. The most important part of my research question addresses how one would go about compiling Agda to Java. Therefore our compilation process is looked at next, as well as its limitations and what subset of the Agda language can be compiled. Then Agda to Java will be compared to existing methods and the viability of the current methodology will be discussed. Afterwards, section 5 will discuss the measures that were taken to ensure the findings were presented in a responsible manner. Then, section 6 will provide my final thoughts and will explain why we do not think Java is a good target language for compilation from Agda. Finally, some future work is discussed and related work will be put into context.

2 BACKGROUND INFORMATION

2.1 Agda

Agda is the language used in this paper, therefore we will give it a brief introduction. More in-depth information is available online[1].

Agda is a functional programming language and differs from languages such as Haskell in its dependently typed nature. Due to a mix of dependent types and functional programming, it is possible to write programs, specifications and proofs all in the same language. In this paper, Agda is mainly used as a programming language and its proof and verification capabilities are only touched upon lightly.

Agda as a language is quite similar to Haskell, but with some major differences. In Agda type declarations are defined similarly to Haskell, but you are allowed to index data types. This means that is possible to encode the length of a list in its type. Right in Figure 1 such an Agda list can be seen.

```
-- Haskell                  -- Agda
data List a =               data Vector (A : Set) : Nat -> Set where
       Nil                    Nil : Vector A zero
    |  Cons a (List a)        Cons : {n : Nat} -> A -> Vector A n -> Vector A (suc n)
```

Fig. 1. "Vector in Haskell and Agda"

This definition looks the same as the List definition in Haskell, where you have the same structure. To define a list, you have your base case **Nil**, which constructs an empty Vector, and the **Cons** case constructing a Vector whose length is one larger than the previous one.

We can take advantage of this more expressive data type by defining a safe head function where its type ensures it will only be called on non-empty lists, an example can be seen below in Figure 2.

```
-- Haskell
head :: [a] -> a              -- Agda
head [] = error "Empty!"      head : {A : Set} {n : Nat} -> Vector A (suc n) -> A
head (x:xs) = x               head (x :: xs) = x
```

Fig. 2. "Normal head function in Haskell and total head function in Agda"

As a matter of fact, it is actually impossible in Agda to define the traditional non-total head function. This is because Agda requires that all functions are total, which gives them additional safety features. This is because Agda needs these requirements to be a proof checker and non-total functions would make Agda's logic inconsistent. Agda also ensures all functions terminate tanks to

[1] https://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials

its termination checker Foetus [Abel, 1998] and due to these strict requirements, will never throw runtime errors.

2.2 Existing Back-ends

The most prominent back-end Agda has is the MAlonzo back-end [Benke, 2007]. The MAlonzo back-end targets Haskell and uses the Glasgow Haskell Compiler (GHC) to create executables from the generated Haskell code. MAlonzo compiles the Agda language quite well, but it has some limitations. Since Haskell is not a dependently typed language, MAlonzo cannot use dependent types and thus uses a workaround. What MAlonzo does, is insert unsafe type coercions. Due to these type coercions, MAlonzo generates a lot of code to do simple things in Agda.

These coercions have a considerable performance impact because they prevent GHC from optimizing type-directed optimizations as well as the blowup in the size of the code. Even though these coercions impact performance, they do not impact the correctness of Agda programs, since they were already type-checked by Agda itself.[Hausmann, 2015, p. 7]

Besides the MAlonzo back-end, there are a variety of other, more experimental back-ends. These back-ends target various languages like JavaScript [Team, 2021], the Epic compiler[Frederiksson and Gustafsson, 2011] and UHC [Hausmann, 2015].

2.3 Choosing a Target Language

In a situation where no previous compilers for Agda exist, it would make the most sense to target another functional language like Haskell. This is not the case today however, Agda has a working back-end for Haskell and decent support in a variety of languages. One language type Agda has no back-end for yet are object-oriented languages and we decided to pick Java.

Java is an interesting choice due to the existing Java ecosystem. Being able to prove various parts of this ecosystem and interact with Java functions using foreign function interfaces could be the next step for both Java and Agda development. The reason Java was chosen instead of a functional language that compiles into Java, like Scala [Odersky, 2022] or Clojure [Hickey, 2022], is that we thought that it would be interesting to see if there were new ways of implementing functional features in languages like Java. Targeting Java directly could also make the implementation of a foreign function interface much easier.

2.4 Agda2Java

Agda2Java, our Agda to Java compiler, was written in Haskell and can be found here[2]. This is because Haskell provides functionality that breaks down an Agda program and converts it to Agda's treeless syntax. This treeless syntax is the output of the Agda typechecker and therefore most types have been erased and is a simplified version of the original Agda program. This is very nice when implementing a back-end because it reduces the amount of features your compiler has to support.

3 TRANSLATING AGDA TO JAVA

Looking at Agda and Java code side by side, it is immediately obvious there are vast differences between the two languages. Differences between the type systems that are used, differences between the paradigms used, functional vs object-oriented programming, and differences between evaluation types. These differences between the two languages are the main hurdle in translating Agda to Java, hence this dedicated section.

This section discusses some of the main problems in the translation process and shows how these problems were solved. The limitations of our current approach are also discussed briefly.

[2]https://github.com/LukasZim/agda2java

3.1 Preamble

For every Agda program the preamble is prepended to the generated Java file, this preamble allows us the support and simulate many of the features of Agda. The Agda2Java preamble includes multiple classes, but since it makes more sense to explain them whenever they are used, the following Figure 3 only shows the two basic building blocks. `interface Agda {}` is something that could be compared to `Set` in Agda. As will be shown later, all data extends the Agda class and all functions will do too. The `interface Visitor` is something that will ultimately help us implement the visitor pattern that will allow us to do pattern matching on types which Java does not natively do.

```
interface Visitor {}
interface Agda {}
```

Fig. 3. Part of the Preamble

3.2 Data Types

The most basic element of any Agda program is the datatype, which is why it is important to make sure they are translated well, such that no functionality is lost. To do this, let us take a look at the Nat type which represents natural numbers as seen in Figure 4.

```
data Nat : Set where
  zero : Nat
  suc : Nat -> Nat
```

Fig. 4. "Nat in Agda"

The datatype `Nat` is part of `Set`, as all Agda types are, and has two different constructors. The `zero` constructor represents the number zero and takes no arguments and returns a natural number. The `suc` constructor takes a natural number as argument and returns a natural number. This way it is possible to represent the number 0 by creating `zero` and the number 2 by creating `suc` (`suc zero`) which is the same a saying that 2 = 1 + (1 + 0)

The Java equivalent can be seen in Figure 5. There are several reasons this representation was chosen. First of `NatVisitor` is created, which we will talk about more in the pattern match section, so let us ignore it for now. The Datatype `Nat` is constructed by an abstract class, which is then extended by classes which include both constructors. In the case of `zero` which has no arguments, it simply includes an empty constructor, as well as the match function which is also part of the pattern matching and which will be discussed there. In case of a constructor with multiple arguments, like in the `suc` case, an additional field is added to the class which is assigned to the value given in the constructor.

An avid reader will have noticed that the class implementing the data type `Nat` extends a class called `AgdaData`, which is the Java equivalent of `Set`. In the next section we will show how this `AgdaData`, together with the Visitor interface, will allow us to do pattern matching. `AgdaData` is shown in Figure 6.

3.3 Pattern Matching

One of the main things Java lacks, compared to functional languages is proper pattern matching. Currently, Java does support some pattern matching, it is possible to pattern match on literals such as Integers and Strings. Java's current pattern matching however, is nothing more than a glorified

```
interface NatVisitor extends Visitor {
    Agda suc (Agda arg1);
    Agda zero ();
}
abstract static class Nat implements AgdaData {}
static class suc extends Nat {
    private final Agda arg1;
    public suc (Agda arg1) {
        this.arg1 = arg1;
    }
    public Agda match (Visitor visitor) {
        return ((NatVisitor) visitor).suc(arg1);
    }
}
static class zero extends Nat {
    public zero () {
    }
    public Agda match (Visitor visitor) {
        return ((NatVisitor) visitor).zero();
    }
}
```

Fig. 5. "Nat in Java"

```
interface AgdaData extends Agda {
    Agda match(Visitor visitor);
}
```

Fig. 6. "AgdaData from the preamble"

switch statement that is not capable of distinguishing anything more accurate than Java literals. For our needs, it needs to be possible to distinguish different types of user-created objects.

A function Definition of the Not function which uses pattern matching can be seen in Figure 7 , which matches on a constructor and returns a corresponding object. The function `minusOne` is also available in Figure 7 and shows a function that tries to subtract 1 from a natural number unless it is the number zero.

```
minusOne : Nat -> Nat
minusOne zero = zero
minusOne (suc n) = n
```

Fig. 7. "Pattern matching in Agda"

The most obvious way of implementing pattern matching in Java would be to create an if-else list of **`instanceof`** statements that ultimately tell you what kind of object you are dealing with. This approach would work for most examples and could be relatively efficient, but it lacks structure. For a proper implementation of pattern matching, which does support dependent types and does support complex objects a more structured approach is desired.

The answer, in this case, is the Visitor Pattern. The Visitor Pattern is a design pattern for object-oriented languages that allows the programmer to define new operations on a certain data structure without changing the underlying classes of the Objects. The drawback of the visitor pattern is that classes cannot be created on the go and that all classes must have a corresponding match function[Palsberg and Jay, 2007]. Since we have control over how we generate the Java classes, we can generate code in such a way that both these drawbacks will not be problematic. The Visitor Pattern can be used to simulate pattern matching in object-oriented languages[Peterson, 2015]. as can be seen in Figure 8 where Java code was generated that does the same thing as the Agda function `minusOne` in Figure 7.

```
var minusOne = (AgdaLambda) x -> {
    return ((AgdaData) x).match(new NatVisitor() {
        @Override
        public Agda zero() {
            return new zero();
        }

        @Override
        public Agda suc(Agda arg0) {
            return arg0;
        }
});};
```

Fig. 8. "Pattern matching in Java"

To implement the `minusOne` function, first a `AgdaLambda` has to be created. `AgdaLambda` is the part of the preamble that represents an Agda function in Java and its implementation can be seen in Figure 9. Another thing to note is that the Java function `minusOne` is a variable and will be inside of another method, this allows us to use functions such as `minusOne` as first-class citizens.

```
interface AgdaLambda extends Agda {
    Agda run(Agda arg);
}
```

Fig. 9. "Java Preamble part 3"

To implement pattern matching, various things need to be done. First, the function input `x`, which is of type `Agda`, needs to be cast to `AgdaData`, which is the part of the preamble that represents Agda datatypes.

Then x's `match` function, which we glossed over in the previous section, is called. What the `match` function does, is it takes an implementation of the Visitor interface and calls the visitors function that corresponds with the constructor it finds. So in case `(new zero()).match(someVisitor)` is called, the function `zero()` will be called inside the visitor and vice versa with `(new suc(someNat)).match(someVisitor)` and the `suc(someNat)` function.

This `match` function then takes an implementation of a Visitor, which has to have a certain amount of methods. In our case, we always have as many methods inside the Visitor implementation of a datatype, as we have constructors. In case of **`Nat`**, **`NatVisitor`** has two methods, one for `zero` and one for `suc`. This way it is possible to simulate pattern matching on datatypes in Java. In case of minusOne, whenever the `zero()` method is called inside of the **`NatVisitor`**, another `zero` is

returned and whenever `suc(arg0)` is called `arg0` is returned. This works in cases where we know we only match on one certain data type, however, sometimes you need to match on an input that might be of a variety of types. That is why the `match` function takes a generic Vistitor instead of a **`NatVisitor`**. So for every separate function that needs pattern matching, a new implementation of **`NatVisitor`** can be found. In cases where matching on multiple types is needed, it is possible to create a custom Visitor just for that function. This however is only something that can be done in theory and has not been implemented into the current version of Agda2Java.

In general this approach of simulating pattern matching works, but has some limitations. The main limitation of the current approach is that for every pattern match, all alternative pattern matches have to be constructed. This is troublesome if you want to pattern match on a very specific case, lets say the number 20, and do something else in every other case. With the current implementation, every `zero` and `suc(arg0)` case would have to be constructed, even if they all compute to the same thing. Another limitation of the current approach is speed and memory usage, as will become visible in section 4 Agda2Java is very inefficient when doing deep recursions. The current approach requires that for every function call, multiple function calls are needed and that the Visitor object is instantiated. This very quickly takes up large amounts of memory and causes stack overflows quicker than really necessary.

3.4 (Partial) Function Application

One important feature of functional programming languages is partial function application. This allows programmers to partially apply functions and pass around those partially applied functions as new functions of their own. Imagine a function sub that takes two arguments, x and y, and subtracts y from x. This can be represented like: `sub(y,x) = x - y`. From sub we could then construct the function `minusOne` where we partially apply the number 1 for y. `minusOne = sub(1,x) = x - 1`.

An implementation of `minusOne` in Agda can be seen in Figure 10. Here we define a function `minus` that subtracts the first argument from the second. Then we define the function `minusOne'`, where we partially apply one as the first argument of `minus`. Finnaly we call `minusOne'` with the value `two`, this result is stored in `ans` and of course equals one, since `2 - 1 = 1`.

```
minus : Nat -> Nat -> Nat
minus zero x = x
minus y zero = zero
minus (suc y) (suc x) = minus y x

minusOne' : Nat -> Nat
minusOne' = minus (suc zero)

ans = minusOne' (suc (suc zero))
```

Fig. 10. "Partial function application in Agda"

Java has some built-in support for partial function application for normal Java functions, but it is very inflexible. Luckily in Java 8 support was added for lambda expressions, these allow us to more easily do partial function application and allow us to pass functions around as first-class citizens. To further help with implementing function application we need the last part of the Java preamble which helps with function application and can be seen in Figure 11. The method `runFunction` takes an argument and a lambda and returns the result of applying that argument to the lambda function.

```
        public static Agda runFunction (Agda arg, AgdaLambda l){
            return l.run(arg);
        }
```

Fig. 11. "RunFunction from the Java preamble"

```
private static AgdaLambda minus;
minus = (AgdaLambda) x -> (AgdaLambda) y -> {
            return ((AgdaData) x).match(new NatVisitor() {
                @Override
                public Agda zero() {
                    return y;
                }

                @Override
                public Agda suc(Agda arg0) {
                    return ((AgdaData) y).match(new NatVisitor() {
                        @Override
                        public Agda zero() {
                            return new zero();
                        }

                        @Override
                        public Agda suc(Agda arg01) {
                            return runFunction(arg01,
                                (AgdaLambda)runFunction(arg0, minus)
    );}});}});};

var minusOneAlt = (AgdaLambda) runFunction(new suc(new zero()), minus);

var ans = runFunction(new suc(new suc(new zero()))), minusOneAlt);
```

Fig. 12. "Java (partial) function application"

The Java implementation of `minus` can be seen in Figure 12. Some things about the generated Java code might seem strange, but there is a reason for most of them. First of all, the reason that `minus` is first declared and then assigned, is because `minus` is a recursive function. If we were to assign and declare `minus` in the same line, Java would throw an error when referencing `minus` in the recursive call. `minus` itself does some pattern matching as explained in the previous section and returns the value `y - x`.

In `minusOneAlt`, a single argument is passed to the function `minus` which returns a function that takes one argument. In our implementation, all multi-argument functions are curried such that you can only apply a single argument at once. This way partial application needs to be possible to support multi-argument functions and it simplifies our definition of the `runFunction` method.

And to finish the function application, a single argument is passed to the newly created `minusOneAlt` function and the result is stored inside `ans`.

4 EXPERIMENTAL SETUP AND RESULTS

To answer our research question: *"Under which conditions is Java a suitable target language for extraction from Agda compared to existing methods?"*, the compiler was benchmarked. This was done by testing multiple Agda files with different input sizes. These exact same tests were also conducted with the existing MAlonzo back-end and the Chez Scheme back-end created by Jesper Cockx [3]. These two back-ends were chosen because we wanted to compare with the state-of-the-art Agda MAlonzo back-end, as well as the more experimental Scheme back-end.

Whenever run times are shown, run times with small input sizes should be taken with a grain of salt and not compared between languages. This is because Java, Haskell and Scheme all have different ways of getting input from the console and these IO methods all have different execution times. Especially for small inputs this IO overhead may take priority over the execution time of the feature that was tested. In some cases the output of the Java back-end had to be manually adjusted because some bugs caused functions to pattern match on the wrong variables. In some other cases, the generated code was adjusted slightly to make it easier to benchmark the programs. None of these changes however impact the execution of the feature that was tested and only change the overhead slightly.

To measure execution times, a command called `Measure-Command {command to be tested}` was used which shows execution times for commands run in Windows PowerShell. The text `"command to be tested"` was changed with a command that runs code for its respective back-end. Generated Java files were first compiled and then run with the command `java -Xss999M -Xmn999M Main` which allows Java to be run with a larger stack and heap. Generated Haskell code was also first compiled, Haskell code however generated an executable that could simply be run with `./executableName`. Scheme code however was interpreted with Petite Chez Scheme 32-bit because the normal Chez Scheme environment was not available for Windows from the official website[4]. Scheme was run using `petite schemeFileName.ss`

4.1 Consuming Numbers

In the first test case the function consume is looked at. The Agda implementation of this function can be seen in Figure 13 and what this function does is, it recursively calls itself on a Natural number, each time subtracting one from that number until the number is zero.

```
consume : Nat -> Nat
consume zero = zero
consume (suc n) = consume n
```

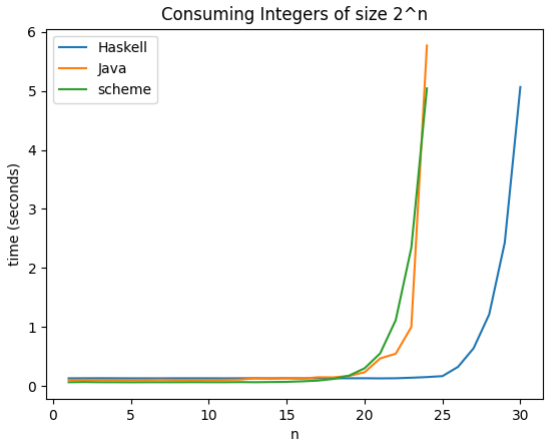
Fig. 13. "The Agda version of the Consume Function"

The interesting thing about this function is that, if no tail call optimization is used, the function will create a very large stack and heap. This is actually the case with Java, which has no tail call optimization. The only way for Java to execute this program with large inputs, is by manually telling Java to use the maximum stack and heap size of one gigabyte and in that case it will use up to eleven Gigabytes of memory on my machine. It is actually not possible for me to run this in Java with an input size of 25 or larger which causes Java to throw a stack overflow error.

It might seem like Scheme is struggling as much as Java is when you look at Figure 14 above which shows a graph relating input size with execution time. The big difference between Java and scheme however, is that scheme uses less than a gigabyte of memory at worst, while Java

[3]https://github.com/jespercockx/agda2scheme
[4]https://scheme.com/download/

Fig. 14. Results of running the Agda function Consume with different Back-ends



uses all memory available on the machine for large inputs. This means that at a certain point Java will simply run out of memory and have to give up whereas Scheme will keep executing. The reason scheme is able to handle larger numbers than Java is because Scheme does have tail-call optimization, which helps a lot in this case. Another difference between Java and scheme is that Java code is (partially) compiled and Scheme is interpreted. This usually means that the interpreted language has a performance drop, when comparing to a compiled language. Yet Scheme seems to execute at the same pace as Java.

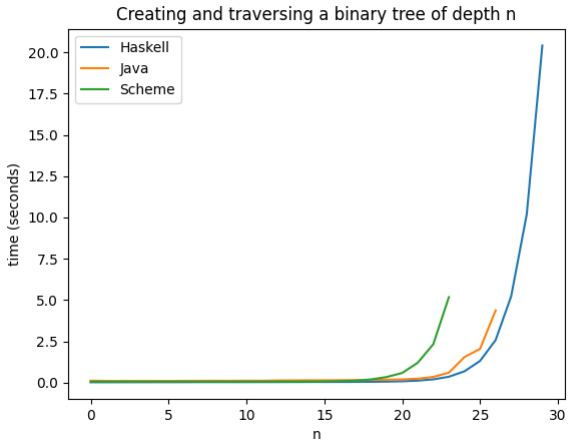
Looking at Figure 14 again, there is a large performance gap between Haskell and either Java or Scheme. This example is a case which suits Haskell perfectly, it can make use of its tail call optimization and its fast recursive function calls, which give it this massive performance boost compared to the other languages. The main advantage Haskell has over a language like Scheme is its compiled nature and its many optimizations. All of these factors contribute to Haskell's fast execution times.

4.2 Traversing a binary tree

In the second test case, each back-end will create and traverse a balanced binary tree of depth n. A balanced binary tree of depth n has 2^n elements in it. This will test how well each back-end performs in cases where tail-call optimization is less important. What is important for this test case is how much memory is available during execution and how efficient is it used as well as how fast the trees can be traversed. In Java's case, a stack overflow will not occur since the execution depth is at most n, but 2^n elements will be stored in the heap so at a certain point it will run out of memory.

In Figure 15 the execution times of the tree test case can be found. The first interesting thing is how fast the Scheme back-end gives up on the execution of the tree test case. Here Java is capable of traversing 2^4 times as many sub-trees as Scheme in the same amount of time. The reason the Scheme back-end stops its execution this early is because it runs out of memory for inputs larger than 22. During the execution of the Scheme back-end we kept track of how much memory it was using and Scheme never used more than 2 gigabytes of memory. In test cases where more than 2 gigabytes were needed, Scheme throw out of memory errors. We assume this is the case because

Fig. 15. Results of building and traversing a binary tree with different Back-ends



the Petite Chez Scheme interpreter we used was 32-bit and therefore not capable of allocating more than 2 to 4 gigabytes of memory. Another interesting observation is that the Java back-end stops executing for inputs larger than 26. This is because Java too run out of memory for larger inputs. Therefore it seems like the Java back-end seems to be a better fit than Scheme for computational tasks that use a lot of memory, but keep a small stack size.

The Haskell back-end however seems to be in a league of its own, during execution it never used more than 70 megabytes of memory, while still having the fastest execution times. Haskell CPU utilization never exceeded 10% and it still executed faster than Java which would use up to 90% of available processing power.

4.3 Dependent Types

The Java preamble, created to help run the Agda code, encodes all functions as variables by letting them be instances of the `AgdaLambda` class using lambda expressions. This allows us to pass around functions as we please. Another thing the preamble does, is implement pattern matching using objects that extends the Visitor interface. This in turn allows our functions to pass around types as we please and pattern match on types that are passed to the function. One example using dependent types can be found in Figure 16. Here a type U, representing possible types, is a type representing possible input types. The Universe function is a function that turns these representations of types into actual types. And the function f pattern matches on the input type and input value and possibly does some computation on this input value. Due to some mistranslations in the pattern matching, where Agda2Java fails to properly keep track of what it is pattern matching on, this function does not compile out of the box. But by manually changing these small errors, which are due to pattern matching order and not because of dependent types, dependent types can be used freely in Java. The manually adjusted Java code can be found in the Agda2Java repository [5]. This translated file is able to run all three test cases and gives the same output as the Haskell back-end.

[5]https://github.com/LukasZim/agda2java/blob/master/fixedUniverseFile.java

```
data U : Set where
    natType : U
    boolType : U
    listType : U -> U

Universe : U -> Set
Universe natType = Nat
Universe boolType = Bool
Universe (listType t) = List (Universe t)

f : (c : U) -> Universe c -> Universe c
f natType zero = zero
f natType (suc x) = x
f boolType False = True
f boolType True = False
f (listType c) nil = nil
f (listType c) (cons x xs) = (cons (f c x) (f (listType c) xs))

test = f natType zero
test2 = f (listType boolType) (cons False nil)
test3 = f (listType (listType boolType))
          (cons
              (cons False nil)
              (cons nil (cons (cons True nil) nil))
          )
```

Fig. 16. "Functions using dependent types"

5 RESPONSIBLE RESEARCH

This paper describes the implementation of the Agda2Java compiler, as well as challenges that were overcome during its implementation. Agda2Java takes Agda code as input and returns corresponding Java code, that should give the same result. Due to this, there is no problem with bias. The implementation of Agda2Java is available online as an open source project in the public domain. This allows others to see the exact implementation and use it for their own research purposes. Relying on the current implementation of Agda2Java as a guideline for Agda features might be problematic, since not the entire Agda language can be translated. This should however not be a problem since it is stated in this report that Agda2Java has limitations in the code it can produce, as well as the language features that are supported.

Another thing that might be problematic in my research is the fact that in some cases the output of Agda2Java has been manually altered to fix certain bugs in the generated code. Care was given while doing this to not alter the complexity of the generated code and to ensure no code was produced that Agda2Java would not be able to generate. If a new type of compiler from Agda to Java where to be build that uses the methods outlined in section 3, but without the bugs in the current approach, then the same results as in section 4 would be found. Due to the nature of Haskell, Scheme and Java, IO operations are done in a different way. This also caused us to uses slightly different Agda files to help with streamlining these IO operations. In test cases with small input sizes these differences might be the key in performance between platforms, but with larger input

sizes these different ways of handling IO operations become negligible. This was also mentioned however and should not be a problem.

6 CONCLUSIONS

We implemented a new back-end for Agda targeting Java. We were able to compile basic Agda programs and execute them at a similar pace to the Scheme back-end and even compiled some examples using dependent types. However, there are a lot of limitations to using Java as a back-end. Java has no build-in pattern matching, laziness or tail-call optimization which forced us to find workarounds for these problems. Pattern matching has been overcome by creating a preamble together with the visitor pattern to help Java use pattern matching. Laziness is something that should be possible to implement [Douence et al., 2009] [Shi et al., 2016] and is an important feature of Agda, but has not been on the priority list. Tail-call optimization is one of the features Java lacks, even though there are libraries that attempt to implement this feature [Keyser, 2019]. Because Java's lack of tail-call optimization, which really holds it back in most of the test cases, we do not recommend to use Java as a target language. In case there is a need to target the Java ecosystem a functional language that compiles to JVM, like Scala or Clojure, would be a better fit. Scala, which does have native pattern matching and tail-call optimization, would allow for faster and less verbose code to be generated. In its current state, Agda2Java is not in a usable state. Performance wise it is slower than Haskell with some problems that do not seem solvable for the current Java language. And as a language targeting the Java ecosystem, Scala just seems to be the better option.

6.1 Future work

The current implementation of Agda2Java still needs a lot of work before one could even think about using it. One of the biggest features Agda2Java is missing is laziness, which prevents certain programs from being able to execute and changes the entire execution order of the program. One such program can be found in Figure 17, the if_then_else function would currently compute both x and y, whereas in a lazy environment only the chosen value would be evaluated. Currently Agda2Java uses Objects to represent number, which is a really inefficient way of doing things. Natural number optimization is a feature that will bring a measureable performance boost. Another thing that Agda2Java currently struggles with is the sanitization of variable and function names, this prevents Agda2Java from running certain programs using special characters which are allowed by Agda, as well as in-line functions.

```
if_then_else_ : Bool -> A -> A -> A
if true  then x else y = x
if false then x else y = y
```

Fig. 17. "A function that would execute differently in a lazy environment compared to a strict environment"

Most of the problems we encountered in Java are also know by the Java development team. There is currently work being done on the possible implementation of real pattern matching in a future version of Java[Bierman, 2022]. According to Brian Goetz, a Java language architect, tail-call optimization is also something Java will eventually support and it "will eventually get done" [Goetz, 2014]. So even though Java, at its current state, is not a good target language for Agda, in the future this might change.

7 RELATED WORK

Agda has had four relatively mature back-ends over the years, MAlonzo, UHC, the JavaScript back-end and the Epic back-end. Most of these, however, have fallen into disarray. The Epic back-end is

no longer maintained and has been removed from official support. The UHC back-end is currently not up-to-date with the most recent version of Agda. The JavaScript back-end is currently officially supported, but still has a lot of bugs. And the most complete, best optimized and best developed way of compiling Agda nowadays, is by using the MAlonzo GHC back-end.

There has been little work on translating Agda to object-oriented languages. The closest thing to compiling Agda to an object-oriented language is ooAgda [ABEL et al., 2017] which attempts to develop a methodology of writing interactive and object-based programs in dependently typed languages. Currently there technically is support for compiling Agda to JVM, by first compiling Agda with the MAlonzo back-end which creates a Haskell file and then using tools like Eta [ETA Team, 2018] or Frege [the Frege Team, 2011] that compile the generated Haskell code to JVM.

The Java language itself is also still being developed, with many interesting features on the way. As stated earlier, pattern matching is a feature which might be coming in a future version of Java [Bierman, 2022] and might change how Java code is written to better match functional languages. Something also stated earlier in this paper is that tail-call optimization is a feature that will eventually make its way into the Java language [Goetz, 2014], which will have a huge impact on optimizing recursive functions in Java.

REFERENCES

Andreas Abel. 1998. foetus – Termination Checker for Simple Functional Programs.

ANDREAS ABEL, STEPHAN ADELSBERGER, and ANTON SETZER. 2017. Interactive programming in Agda – objects and graphical user interfaces. *Journal of Functional Programming* 27 (Feb 2017). https://doi.org/10.1017/s0956796816000319

Marcin Benke. 2007. Alonzo — a compiler for Agda. https://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf

Gavin Bierman. 2022. JEP 427: Pattern Matching for switch (Third Preview). https://openjdk.java.net/jeps/427

Rémi Douence, Xavier Lorca, and Nicolas Loriant. 2009. Lazy composition of representations in Java. *Software Composition* (2009), 55–71. https://doi.org/10.1007/978-3-642-02655-3_6

Sophia Drossopoulou and Susan Eisenbach. 1997. Is the Java Type System Sound? (01 1997).

the ETA Team. 2018. The ETA programming language, a dialect of Haskell on the JVM. https://github.com/typelead/eta

Olle Frederiksson and Daniel Gustafsson. 2011. https://publications.lib.chalmers.se/records/fulltext/146807.pdf

Brian Goetz. 2014. *Brian Goetz - Stewardship: the Sobering Parts.* https://youtu.be/2y5Pv4yN0b0?t=3782

Philipp Hausmann. 2015. *The Agda UHC Backend.* Ph. D. Dissertation. https://studenttheses.uu.nl/bitstream/handle/20.500.12932/28189/MSc_thesis_Agda_UHC_Backend.pdf?sequence=2

Rick Hickey. 2022. https://clojure.org/

Jude Keyser. 2019. TAIL_REC. https://github.com/Judekeyser/tail_rec

Martin Odersky. 2022. The Scala programming language. https://www.scala-lang.org/

J. Palsberg and C.B. Jay. 2007. The essence of the visitor pattern. *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)* (2007). https://doi.org/10.1109/cmpsac.1998.716629

Kevin Peterson. 2015. Pattern matching in Java with the Visitor pattern. https://eng.wealthfront.com/2015/02/11/pattern-matching-in-java-with-visitor/

Jianjun Shi, Weixing Ji, Lulu Zhang, Yujin Gao, Han Zhang, and Duzheng Qing. 2016. Profiling and analysis of object lazy allocation in Java programs. *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)* (2016). https://doi.org/10.1109/snpd.2016.7515964

The Agda Team. 2021. Compilers¶. https://agda.readthedocs.io/en/v2.6.2/tools/compilers.html

the Frege Team. 2011. Frege. https://github.com/Frege/frege

A APPENDIXES

A.1 The entire Java code of the translated Agda program

```
class main {
    private static AgdaLambda minus;

    // helperfunction from the preamble
    public static Agda runFunction(Agda arg, AgdaLambda l) {
        Agda result = l.run(arg);
        return result;
    }

    // the translated Agda program
    public static void main(String[] args){
        minus = (AgdaLambda) x -> (AgdaLambda) y -> {
            return ((AgdaData) x).match(new NatVisitor() {
                @Override
                public Agda zero() {
                    return y;
                }

                @Override
                public Agda suc(Agda arg0) {
                    return ((AgdaData) y).match(new NatVisitor() {
                        @Override
                        public Agda zero() {
                            return new zero();
                        }

                        @Override
                        public Agda suc(Agda arg01) {
                            return runFunction(arg01,
                                    (AgdaLambda)runFunction(arg0, minus)
                            );
                        }
                    });
                }
            });
        };

        var minusOneAlt = (AgdaLambda) runFunction(new suc(new zero()), minus);

        var ans = runFunction(new suc(new suc(new zero())), minusOneAlt);
    }
```

```
    //generated datatypes
    interface BoolVisitor extends Visitor {
        Agda True();
        Agda False();
    }

    abstract static class Bool implements AgdaData{}

    static class True extends Bool {

        @Override
        public Agda match(Visitor visitor) {
            return ((BoolVisitor)visitor).True();
        }
    }
    static class False extends Bool {

        @Override
        public Agda match(Visitor visitor) {
            return ((BoolVisitor)visitor).False();
        }
    }

    interface NatVisitor extends Visitor {
        Agda zero();
        Agda suc(Agda arg0);
    }

    abstract static class Nat implements AgdaData {}

    static class zero extends Nat {
        @Override
        public Agda match(Visitor visitor) {
            return ((NatVisitor)visitor).zero();
        }
    }
    static class suc extends Nat {
        private final Agda arg0;

        public suc(Agda arg0){
            this.arg0 = arg0;
        }

        @Override
        public Agda match(Visitor visitor) {
            return ((NatVisitor)visitor).suc(arg0);
        }
    }
}
```

```java
// java preamble
interface Visitor {}

interface Agda {}

interface AgdaData extends Agda {
    Agda match(Visitor visitor);
}

interface AgdaFunction extends Agda {
    Visitor getFn();
}

interface AgdaLambda extends Agda {
    Agda run(Agda arg);
}

class AgdaFunctionImpl implements AgdaFunction {
    private Visitor fn;

    public AgdaFunctionImpl(Visitor fn){
        this.fn = fn;
    }

    public Visitor getFn(){
        return fn;
    }
}
```

Fig. 18. "The Generated Java Program"