Bridging the Gap between Haskell and Java

Julian Fleischer

June 27, 2013

Abstract

This thesis investigates the possibility of translating interfaces between Haskell and Java libraries. Furthermore a library and a tool are developed for invoking the virtual machine from within Haskell processes and for the automatic translation of Java APIs into Haskell.

In dieser Arbeit wird die Möglichkeit einer Übersetzung von Schnittstellen zwischen Haskell und Java Bibliotheken untersucht. Außerdem werden eine Bibliothek und ein Werkzeug entwickelt um die virtuelle Maschine aus Haskell Prozessen heraus zu nutzen und um automatische Übersetzungen von Java APIs nach Haskell durchzuführen. vorgesetellt

Selbstständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Ausarbeitung mit dem Titel: "Bridging the Gap between Haskell and Java" selbstständig und ohne unerlaubte Hilfe angefertigt habe.

Ich versichere, dass ich ausschließlich die angegebenen Quellen in Anspruch genommen habe. Zitate sind als solche gekennzeichnet, durch Anführungszeichen und Fußnoten.

Julian Fleischer

Contents

1 Introduction and Motivation **5**
1.1 Scope of this Thesis 5
1.2 Organization of this Document 5

2 Similarities and Differences between Haskell & Java **7**
2.1 A Brief Overview of the Haskell Programming Language 7
2.2 A Brief Overview of the Java Programming Language 7
2.3 Syntax and Semantics 8
2.3.1 (Restricted) Polymorphism 8
2.3.2 Higher-kinded Polymorphism 9
2.3.3 Order, State, and Side Effects 10
2.3.4 Excecution Model 10
2.4 The Awkward Squad 12
2.4.1 Exceptions 12
2.4.2 The Java Virtual Machine 13
2.4.3 Haskell Implementations 14
2.5 Differences and Similarities rounded up 14

3 Translating APIs between Haskell and Java **16**
3.1 Translating Java idioms into Haskell 16
3.1.1 The Zoo of Things in Java vs. Functions in Haskell 16
3.1.2 The Java Monad 18
3.1.3 Primitive Types 18
3.1.4 Objects 18
3.1.5 Subtype Polymorphism 19
3.1.6 Static and Non-static Methods 20
3.1.7 Higher Kinded Types and Type Variables 21
3.1.8 Arrays 22
3.1.9 Method Return Types and Exceptions 22
3.1.10 A Note on Return Types and Type Casts 24
3.1.11 Anonymous Classes 24
3.1.12 Summary 26
3.2 Translating Haskell idioms into Java 28
3.2.1 There is only Data 28
3.2.2 Algebraic Datatypes and Constructors 28
3.2.3 Type Classes 29
3.2.4 Polymorphic Return Types 31
3.2.5 Summary 31

4 The java-bridge library **33**
4.1 Prerequisites 33
4.1.1 The Haskell Foreign Function Interface 33
4.1.2 The Java Native Interface 34

4.2 Design of the Library 34
4.3 Low Level Interface 34
4.3.1 IDs and References 35
4.3.2 Controlling the Java Virtual Machine 35
4.3.3 Class, Method and Field Lookup 36
4.3.4 Invoking Methods 37
4.3.5 Getting and Setting Fields 37
4.3.6 Argument Passing 38
4.3.7 Releasing Resources 38
4.3.8 Support for Complex Datatypes 39
4.3.9 Reflection 39
4.3.10 Exception Handling 39
4.3.11 Callbacks 39
4.3.12 Workarounds 40
4.4 Medium Level Interface 41
4.4.1 Haskell extensions being used 41
4.4.2 The Java Monad 42
4.4.3 Method and Field Descriptors 43
4.4.4 References 45
4.4.5 Callbacks 45
4.4.6 Garbage Collection 49
4.4.7 Concurrency 49
4.4.8 OS X Issues 50
4.4.9 Exception Handling 51
4.4.10 Retrieving Classes, Fields, and Methods 51
4.4.11 Calling Methods 52
4.4.12 Reading and Writing Fields 52
4.4.13 Utlity Functions 52
4.4.14 Interaction with the IO Monad 53
4.5 Summary 53

5 The j2hs Bindings Generator 55
5.1 Prerequisites 55
5.2 Design 55
5.2.1 Circular References 55
5.2.2 Dependency Hell 57
5.3 Implementation Details 58
5.3.1 The Support Module Foreign.Java.Bindings 60
5.3.2 coerce 61
5.3.3 Casts 61
5.4 Summary 62

6 Example: A Swing Calculator written in Haskell 63

7 Summary 65

7.1 Related work 65
7.2 Further directions 66
7.3 Where can the java-bridge be found? 66
7.4 Acknowledgements 66

A References **67**

1 Introduction and Motivation

The crazy think[sic!] is we still are extremely bad at fitting things together – still the best way of fitting things together is the Unix pipe
– Joe Armstrong[1]

Haskell and Java are two very different programming languages. Haskell is typically compiled to native binaries, whereas Java is run in a virtual machine. Haskell is a non-strict language with immutable data, whereas Java is a strict language with mutable data. People having an interest in Haskell are mostly from an academic background, the Java community is largely rooted in the industry.

Both languages have their strengths and weaknesses. Haskell is a very well designed language with strong guarantees about the validity of resulting applications. It is however rather complicated to reason about the performance of a particular piece of code, since Haskell is such a powerful abstraction that the underlying machine almost vanishes behind a pure model of computation.

Java is the culmination of decades of engineering in imperative programming with a huge community and a large set of available libraries. The language however is rather bloated. Java is also very roomy, by which is meant that Java allows anything to happen at any time. As an impure language certain actions like writing to `stdout`, or creating a file in the file system can not be prevented from happening.

It would be quite beneficial for both camps to have a means of interfacing with each others language: there are libraries for almost everything written in Java, which one might want to use in Haskell. On the other hand, certain applications are definitely far easier and robust to be written in Haskell (parsers are a canonical example), and it might be desirable to outsource parts of a Java project to Haskell.

1.1 Scope of this Thesis

In this thesis two tools are presented: The *java-bridge* library and the *j2hs* bindings-generator. The java-bridge is a library that allows a Haskell program to invoke methods in the Java Virtual Machine and to call back into the Haskell runtime.

Because of the vast differences between Haskell and Java – especially relating to their type systems – a closer investigation of these differences is conducted and a translation scheme is proposed which captures the essence of a Java API and expresses it in Haskell. The *j2hs* tool performs this translation automatically and uses the java-bridge library to connect the two runtimes.

1.2 Organization of this Document

In section 2 the Haskell and Java programming languages are described and compared. The parts that are especially interesting regarding our goal of creating a tool that automatically creates bindings between applications and libraries written in these two languages are paid attention foremost.

How the interface of a Java library could be translated into Haskell such that it looked like a Haskell library and vice versa is discussed in the next section, section 3. We will see that a Java interface can be translated straightforward to Haskell. On the other hand translating a Haskell interface to Java bears several problems.

Section 4 than explains the design and implementation of the *java-bridge* library, which focuses on technical details like concurrency and garbage collection between, and how the Java Virtual Machine and the Haskell runtime affect each other.

The observations from section 3 and the results from 4 are than combined to build

[1] `http://erlang.org/pipermail/erlang-questions/2013-January/071944.html` – January 2013, Joe Armstrong in an email

the *j2hs* tool. The j2hs tool is used to perform a complete translation of the standard library of the Java Standard Edition (Java SE).

These tools are used in section 6 to demonstrate how a Haskell programmer could use the Java standard library without leaving the Haskell programming language:A GUI application utilizing the Java Swing UI Toolkit (actually a very simple calculator written in Swing).

We conclude with an evaluation of the tools built and compare them to other projects that aim to achieve some kind of interoperability between Haskell and Java.

Citations are markes as such by "..." and footnotes. An exhaustive list of references is maintained in the appendix. To keep the footnotes clear, only the name of a document that the citation is from is mentioned. The exact reference is to be looked up in the appendix.

2 Similarities and Differences between Haskell & Java

We've got ears we've both got eyes
We've got tails to swat the flies
Yes we're different as can be
But in some ways we're the same, you and me
– Buster the Horse (Sesame Street)[2]

2.1 A Brief Overview of the Haskell Programming Language

Haskell is a functional language based on a "slightly sugared variant of the lambda calculus"[3]. Along these sugar parts of Haskell are *algebraic data types* – or *tagged unions*[4] – which allow all data types to be described. In fact even machine level integers may be described by enumerating all integers and fitting them into an algebraic data type. The only control structures known to Haskell are *pattern matching* and *recursion*. These allow for the definition of all higher level structures like `if/then/else` or `case/of`.

Above the core language, which deals with *values*, there is a rather sophisticated type system: "Values and types are not mixed in Haskell. However, the type system allows user-defined datatypes of various sorts, and permits not only *parametric polymorphism* (using a traditional Hindley-Milner type structure) but also *ad hoc polymorphism*, or *overloading* (using *type classes*)"[5].

The type system is probably Haskells most distinctive feature and it is most certainly the very reason for why "[...] Haskell is doomed to succeed" (– Tony Hoare). Many extensions have been proposed to the type system, making it strive towards a dependently typed language[6].

2.2 A Brief Overview of the Java Programming Language

The Java Language Specification describes Java as "a general-purpose, concurrent, class-based, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language"[7].

The ingenious article "A Brief, Incomplete, and Mostly Wrong History of Programming Languages" describes Java as "a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance."[8]

Both of these views hold a certain truth. The core Java language is an imperative programming language that uses well-known control structures such as `if`, and `while` for branching and looping. Values are mostly incarnated as *objects* whose structure is determined by *classes*[9]. Code is strictly organised into these *classes* which in turn may be grouped into *packages*[10]. Besides classes and objects there are eight *primitive types* which can not be defined by means of the language but have to be taken as a given.

It is however debatable whether Java really is "simple". I for one have come to the conclusion that Java is neither simple nor concise. In fact, it is a monster of

[2] `http://www.youtube.com/watch?v=O-9PX95lE8c` – Sesame Street - Different Yet the Same
[3] Haskell 2010 Language Report §1.2 The Haskell Kernel
[4] Hitchhikers guide to Haskell
[5] Haskell 2010 Language Report §1.3 Values and Types
[6] Giving Haskell a Promotion – Introduction
[7] Java Language Specification, Chapter 1. Introduction
[8] `http://james-iry.blogspot.de/2009/05/brief-incomplete-and-mostly-wrong.html` – James Irys Blog – May 7, 2009
[9] Java Language Specification, Chapter 8. Classes
[10] Java Language Specification, Chapter 7. Packages

complexity, requiring the programmer to repeat himself over and over again. We will see that there are a lot of special cases within the Java programming language which has to be dealt with.

2.3 Syntax and Semantics

Both Haskell and Java are meant to be general purpose programming languages. Both of them offer a *managed environment* by which is meant that the state of the underlying machine is more or less hidden and needs not be taken care of by the programmer. It is especially not necessary to deal with the layout of memory or to free unused memory.

Besides these technological similarities, there are notable differences in their syntax and semantics. This section will describe several of these differences, but only in so far as they concern the translation of programming interfaces between Haskell and Java. Therefor features like *type inference* in Haskell contrasted to *manifest typing* in Java will not be covered in this section.

The section after this section will discuss a few technological aspects of the two programming platforms. By "technological" we understand assumptions about the underlying machine and about the actual implementation of Haskell and Java.

2.3.1 (Restricted) Polymorphism

While both languages support some kind of polymorphism they do so in very different manners. Polymorphism in Haskell is achieved via *parametric polymorphism* by which the argument of a function or the field of an algebraic data type can be made to accept any type instead of a specific one:

```haskell
id :: a -> a
id x = x
```

Figure 1: *The polymorphic id function*

```haskell
data List a = Cons a (List a)
            | Nil
```

Figure 2: *A polymorphic List*

Polymorphism in Java on the other hand is handled via *subtype polymorphism*. Everywhere where a value of a certain type is expected, a value of a subtype of that type can be used. Essentially this makes any method or field that takes a non-primitive value polymorphic.

The argument `arg` in the following example may not only be a value of type `Number`, but also of any type that is a subtype of `Number` – such as `BigDecimal` or `AtomicLong`:

```java
class A {
    public void aMethod(Number arg) {
        ...
    }
}
```

Figure 3: *A polymorphic method in Java*

Subtype polymorphism is a form of *restricted polymorphism* since it effectively limits the set of possible types to types for which there is a subtyping relationship between them and the given type.

The polymorphic type of a function in Haskell may be restricted in a similar way. Functions may declare a certain *context* which consists of a set of restrictions that apply on the type variables used in the declaration of the function.

```haskell
(!=) :: Eq a => a -> a -> Bool
```

Figure 4: *This fictitious operator is polymorphic, but 'a' is restricted.*

Such contexts mention *type classes* that are groupings of several types. A type is a member of a specific type class if and only if there exists a so called *instance* of the type for the given type class.

```haskell
data GaussianInteger a = GaussianInteger a a

instance (Eq a, Integral a) => Eq (GaussianInteger a) where
    GaussianInteger x y == GaussianInteger x' y'
        = (x, y) == (x', y')
```

Figure 5: *An instance declaration in Haskell.*

2.3.2 Higher-kinded Polymorphism

Versions 1.0 up to 1.4 of the Java programming language did not feature any other kind of polymorphism but subtyping polymorphism. This changed with the advent of Java 5 (which is the successor to Java 1.4) and the introduction of *generics*. Java generics extend the language with type parameters and effectively introduce parametric polymorphism to the Java programming language[11]. Java generics were devised, among others, by WADLER who was also part of the committee that created Haskell 98 and Haskell 2010.

Generics allow for type parameters associated with classes and methods. These type parameters may be used as type variables in the declaration of fields and methods. Thus it is possible to define polymorphic types just like in the Haskell examples from figures 1 and 2 in Java:

```java
...
    public static
    <A> A id(A a) { return a; }
...
```

Figure 6: *Polymorphic id in Java*

```java
class List<A> {
    public void
    add(A element) { ... }
}
```

Figure 7: *A polymorphic list in Java.*

Both Haskell and Java support multiple type parameters:

```java
class Map<K,V> { ... }
```

Figure 8: *A polymorphic map in Java*

```haskell
data Map k v = ...
```

Figure 9: *The same in Haskell*

It is however not possible to apply a parameter to a type variable in Java whereas in Haskell it is. In Haskell one could for example write a function that deals with "things that contain things", whereas in Java we can not:

```haskell
toList :: Traversable c => c e -> [e]
```

Figure 10: *A higher kinded type variable in Haskell:* $c :: * \to *$

[11]Generic Java: Extending the Java programming language with type parameters

```
public <C extends Traversable,E>
List<E> toList(C<E> c) { ... }
```

Figure 11: *This is not possible in Java: E is applied to a type variable.*

2.3.3 Order, State, and Side Effects

Nothing endures but change.
– Heraclitus

The key difference between a Java program and a Haskell program is that a Java program is a sequence of method calls whereas a Haskell program is a set of functions. One consequence of this is that the order of execution matters in a Java program but not in Haskell. Another consequence of this is that in Haskell there is no mutability, i.e. once a name is bound to a value it stays that way. In Java on the other hand, a value associated with a given name may change during the execution of a program due to the side effect of an expression, i.e. a piece of code may re-assign a new value to a variable.

While the imperative way of doing things surely is a very intuitive one, it also tends to lead to a level of complexity that is rather hard to manage. In order to understand the possible states of a given program at a given location in its code, one needs to think about all the possible states that each variable in the program could have at this very location.

In order to cope with this level of complexity things are encapsulated in classes and objects in Java. In fact, the whole point of object orientation in Java revolves around mastering state.

Haskell on the other hand does not have such notion of state nor could a function possibly have a side effect that affected the global state of the program – simply because there is none. This is a necessary consequence from the fact that a Haskell program is an unordered set of functions.

While this might seem unintuitive at first or even inefficient it is not. Since restrictions on global state and sequential order haven been lifted, a compiler might perform optimizations that it simply was not allowed to do in an imperative setting.

It is however possible to simulate sequencing, state, and side effects in Haskell. The key idea here is that a Haskell program creates a list of actions which are then executed one after another by the runtime system. These actions themselves are typically functions which go from a state to a certain action and a new state: `State -> (Action, State)`.

Since Haskell is a strongly typed language, it is possible to exactly tell what kind of actions a program will produce and what not.

2.3.4 Excecution Model

Efficiency is intelligent laziness.
– Unknown, but accredited to David Dunham

Java is a *strict* programming language, whereas Haskell is a *non-strict* programming language. This is about *reduction* happening from the inside out in a strict language and from the outside in in a non-strict language. We will consider a few examples to grasp these concepts.

```
1 0*(3+(47*9))
2 (0*)(3+(47*9))
3 0
```

Figure 12: *Non-strict reduction*

```
1 0*(3+(47*9))
2 0*(3+((47*)9))
3 0*(3+423)
4 0*((3+)423)
5 0*426
6 (0*)426
7 0
```

Figure 13: *Strict reduction*

In figure 12 a non-strict reduction is taking place. In the first step the `times` function is applied to its first (left) argument, yielding a new function `zero times`. Zero times always returns zero and can thus discard its argument immediately, yielding 0.

In figure 13 a strict reduction can be seen. In the first step the innermost function is applied to its first argument, yielding a new function `47 times`. This happens several times until finally `(0*)426` is reduced, yielding also 0. It can be seen that in this case the non-strict strategy is clearly taking less steps than the strict one.

It is however an unfair example, since Java is not based on *term rewriting.* Also Java does not support the partial application of functions, thus it would transit from line 1 to 3 directly, as from line 3 to 5. But it is a good example, since it well illustrates the difference between strict and non-strict reduction. As Java is not based on term rewriting we will be speaking of *evaluation* instead. A better translation of the example would be:

```
PUSH 9
PUSH 47
TIMES
PUSH 3
ADD
PUSH 0
TIMES
```

Figure 14: *The calculation from above evaluated inside the Java Virtual Machine.*

Figure 14 also takes into account that Java is executed in a virtual machine, which itself is stack based.

Non-Strictness and bottom (⊥) Let us consider a more interesting example:

```
0*(3+(5/0))
(0*)(3+(5/0))
0
```

Figure 15: *Success.*

```
0*(3+(5/0))
0*(3+((5/)0))
error: div by zero
```

Figure 16: *fail!*

```
PUSH 0
PUSH 5
DIVIDE
error: div by zero
```

Figure 17: *fail!*

What is actually interesting about the evaluation strategy is that there is a semantic difference between the two. Since strict evaluation will look at everything and force its evaluation it will also hit every erroneous computation contained in the expression tree, whereas it may happen that a non-strict evaluation entirely skips erroneous parts and delivers a result for an otherwise broken piece of code.

It is often heard that Haskell was a *lazy* programming language. While most implementations of Haskell are in fact lazy by default the statement is not correct per se. Haskell is only non-strict which means that it has to fulfill the semantics discussed above. It would be perfectly valid for a Haskell compiler to compute possibly unneeded results and throw them away, just as Java would. GHC for example features *speculative evaluation* and there is *Eager Haskell*, an implementation of Haskell that is not lazy at all. There are also parts of the Haskell language that are strict by default, such as pattern matching[12] or strict fields[13].

Infinite lists Another interesting property about non-strict evaluation are infinite lists. Since non-strict evaluation will not need to look at all the elements in a list it may skip a possibly infinite computation. Again, a computation which would result in the $\perp$ value using strict evaluation (in the previous example it was an erroneous computation) may result in a non-bottom value using non-strict evaluation:

```
head [1..] -- first element of the infinite list of 1,2,3,...
-> 1
```

Figure 18: *Here, non-strict reduction yields something useful from an infinite list*

2.4 The Awkward Squad

To write programs that are useful as well as beautiful,
the programmer must, in the end, confront the Awkward Squad.
– Simon Peyton Jones[14]

Besides *pure computations* there will always be situations that can not be handled beautifully, so we need to find a way to do it at least gracefully. The Awkward Squad is a term for the inherent relationship of any meaningful program with the real world, which deals with *exceptions*, *concurrency*, *I/O*, and interacting with other kids in the schoolyard.

2.4.1 Exceptions

Exceptions are defined error conditions that may occur during the execution of a program. Some languages like C do not have special support for those conditions but use special return values or state to signal exceptional conditions, other languages like Java have a dedicated data type and syntactic support for raising and handling exceptions.

Java is rather famous for its implementation of exceptions. The language has a feature called *checked exceptions*[15] which forces the programmer to annotate methods with possible exceptions (i.e. declare everything that could possibly go wrong). When such a method is called these possible exceptions must be taken care of, or the Java compiler will refuse to do its job. It would be fair to say that exceptions are ubiquitous in Java, and an integral part of the language.

```
class Exceptional {
    public void method() throws IOException {
        ...
    }
}
```

Figure 19: *A Java method declares that it throws an IOException.*

It would be a very nice property of the language if the absence of such declarations would actually prove the absence of exceptional behavior (except maybe for non-

[12]Haskell 2010 Language Report § 3.17 Pattern Matching
[13]Haskell 2010 Language Report § 4.2.1 Algebraic Datatype Declarations / Strictness Flags
[14]Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell
[15]Java Language Specification § 11.2 Compile-Time Checking of Exceptions

termination). Sadly this is not the case. Besides checked exceptions there are so-called *unchecked exceptions* or *runtime exceptions* that may occur all over the place.

Besides checked and unchecked exceptions there are also errors which in general denote some fatal condition and should not be handled at all:

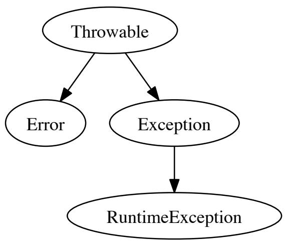


Figure 20: *The Java exception hierarchy*

Haskell on the other hand is widely believed to not have exceptions at all and that there would be no means to deal with exceptional behavior. This is not the case. There are exceptions and there are even means to handle exceptions, known as *IO errors*[16].

The confusion results from the distinction between IO errors, exceptions, and the general notion of *errors*: "Errors during expression evaluation, denoted by $\perp$ (*bottom*), are indistinguishable by a Haskell program from non-termination. Since Haskell is a non-strict language, all Haskell types include $\perp$. That is, a value of any type may be bound to a computation that, when demanded, results in an error. When evaluated, errors cause immediate program termination and cannot be caught by the user."[17]

The name "IO error" stems from the fact that dealing with exceptions is only possible within the IO Monad in Haskell 98 and Haskell 2010. Monads in general provide a `fail` function which allows for customized exception handling and gives rise to solutions such as the *error monad.*

Further means of dealing with exceptions are supported in the presence of Concurrent Haskell[18][19] and implemented in newer versions of GHC[20].

Exceptions in Haskell are admittedly confusing, let alone because there are so many different ways to deal with them.

2.4.2 The Java Virtual Machine

The Java platform not only defines the Java programming language but also the *Java virtual machine* (JVM) which has its own assembler language known as *bytecode.* The behavior of this machine is rigorously defined and covers not only the sequential execution of a Java program but also concurrent execution and the semantics of concurrent memory access. The JVM furthermore defines error conditions such as memory shortage, overflows, or index operations on arrays that are out of bounds. The JVM also does have explicitly defined memory areas such as a *stack* and a *heap* as well as a garbage collector that takes care of managing unused space in the virtual machine.

The Java Native Interface The JVM can be accessed from the outside via the *Java Native Interface* (JNI). The JNI defines a C and C++ interface for interoperating with the virtual machine from native code.

[16]Haskell 2010 Language Report § 42.3 Throwing and catching I/O errors
[17]Haskell 2010 Language Report § 3.1 Errors
[18]Concurrent Haskell
[19]Asynchronous exceptions in Haskell
[20]A semantics for imprecise exceptions

2.4.3 Haskell Implementations

Haskell is defined by the semantics of the underlying lambda calculus, which does not cover a machine level representation of Haskell code. Very many aspects of Haskell are implementation specific (for example the `error` function throws an exception in GHC, which can be recovered from, whereas the language standard only requires it to be a $\perp$ value).

The Foreign Function Interface Haskell 2010 is the successor to the Haskell 98 language standard which incorporates the so called *foreign function interface* (FFI). The FFI "enables (1) to describe in Haskell the interface to foreign language functionality and (2) to use from foreign code Haskell routines. [. . .] The Haskell FFI currently only specifies the interaction between Haskell code and foreign code that follows the C calling convention".

Concurrent Haskell While the FFI addends certain assumptions about Haskell implementations (it for example explicitly mentions a *storage manager* and specifies *pointers* in Haskell), a "major omission is the definition of the interaction with multithreading in the foreign language and, in particular, the treatment of thread-local state, and so these details are currently implementation-defined"[21].

An extension to the Haskell language called "Concurrent Haskell" has been proposed by PEYTON JONES ET AL as well as the extension of the Haskell Foreign Function Interface with Concurrency [22]

The Glasgow Haskell Compilation System The above mentioned extensions to the Haskell language are implemented in the Glasgow Haskell Compilation System (GHC) and I expect them to be incorporated into future revisions of the language standard (such as the FFI already has become part of Haskell 2010).

2.5 Differences and Similarities rounded up

We have seen that things are handled quite differently in Java and Haskell, even antithetic in some key aspects. The most striking differences are:

- strict vs. non-strict semantics – resulting in eager or lazy evaluation by default, and handling some possibly erroneous situations differently;
- mutable data vs. immutable data – once bound the association of a name and a value are not going to change. This is like fixing every variable with a `final` modifier in Java;
- unrestricted side effects vs. restricted side effects – in principle everything is allowed to happen at all times in a Java program, whereas side effects in Haskell are controlled by the type system, and thus certain actions can be prevented from happening (for example writing to `stdout` can not happen in a pure function);
- objects vs. functions as first class citizens – in Haskell everything can be regarded as a function, whereas in Java everything (except for primitive types) is an object;
- polymorphism is handled vastly differently: While Java emphasizes polymorphism by subtyping and allows for overloading of methods by the types of their arguments, Haskell features type classes which allow for both overloading as well as some kind of polymorphism which is able to simulate subtyping (we will see this later on);

[21] Haskell 2010 Language Report § 8.1 Foreign Languages

[22] MARLOW, PEYTON JONES, THALLER Extending the Haskell Foreign Function Interface with Concurrency

Besides these, to some extent drastic, differences, both languages have in fact very much in common:

- As opposed to dynamic languages both languages *have* a type system and types and values are strictly separated (again as opposed to dependently typed programming languages where the type of an expression may depend on a specific value);
- both languages feature a system for throwing and catching exceptions;
- both langauges allow for structured programming, where in Java code is organized in classes, code is organized in modules in Haskell;
- although Java code is interpreted inside the virtual machine, both languages are compiled into some sort of object code. Both languages are, up to a certain extent, platform indepent (though you can not distribute compiled haskell code across different platforms);
- both languages have a notion of parametric polymoprhism (though Haskell has no notion of higher kinded type variables);
- both languages have a defined interface to communicate with programms written in C.

3 Translating APIs between Haskell and Java

There are two sides to every issue:
one side is right and the other is wrong,
but the middle is always evil.
– Ayn Rand

3.1 Translating Java idioms into Haskell

The public API of a Java library consists of a set of *packages*. A package contains a set of classes where a class maybe an actual *public class*, an *interface*, an *enum* or an *annotation*. A Java class declares a public programming interface. In order to avoid ambiguity we will call such an interface a *Java class interface*. We will refer to the set of Java classes in a Java package as *Java package interface*.

Translating a Java class interface to Haskell requires finding a suitable representation of the Java idioms defined above in the Haskell language. We assert that the reader is familiar with the Java programming language and its constructs.

3.1.1 The Zoo of Things in Java vs. Functions in Haskell

Java is a rather complex language contrasted to Haskell. While in Haskell there is only data and a special datatype of kind `* -> *` (functions), Java has fields, functions, constructors, static initializers, behavioural declarations like `strictfp` or `synchronized`, etc. To be fair: There are a lot of different things in Haskell too, but on the surface level they are all types (having a certain kind) or values (having a certain type). This top level space is much more crowded in Java.

When translating these top level constructs from Java to Haskell we have no choice but to turn anything into a function, which especially poses a problem regarding the names of things. While it is possible in Java for two members to have the same name they are not allowed to have the same name in the top level namespace of a Haskell module. Also field access will have to be realized by using a *getter* and a *setter* function. Even worse: Method names in Java are allowed to be overloaded.

While overloading is generally possible in Haskell, it is not the most advisable thing to do. In many situations it will also require the use of certain non-standard Haskell extensions, most prominently *flexible instances*:

```haskell
class MethodPrint a where
    print :: a

-- standard Haskell only allows types to be parameterized
-- by a type variable in the instance head, but (->) is
-- actually parameterized by ‘Int’ and ‘IO ()’ here.
instance MethodPrint (Int -> IO ()) where
    print x = ...

instance MethodPrint (Object -> Int -> IO String) where
    print o x = ...
```

Figure 21: *A possible overloading scheme using flexible instances.*

As a matter of fact, the generous use of overloading will defeat many other features of Haskell, like type inference and composability. Too much polymorphism may also lead to ambiguity in expressions and force the Haskell programmer to annotate many more type declarations than necessary:

There is another issue with overloading: It is typically achieved by using type classes. But as we will see type classes are our weapon of choice for realizing subtype poly-

```haskell
class A a where f :: a
class B a where g :: a -> String

h = g f -- What is the type of f?
```

Figure 22: *Composability screwed up.*

morphism in Haskell. Mixing both the ordinary overloading of methods with fields, constructors, and subtype annotations will definitely pollute any sensible translation with overly verbose type annotations. Adding an encoding for subtype polymorphism will most certainly lead to overlapping instances and eventually render a translation impossible or force an insane amount of type trickery.

We therefore keep things separated. Fortunately the allowable characters for the definition of method names in Java and function names in Haskell are different, so that we can use some special characters to differentiate matters. Unfortunately the allowable characters for the definition of method names in Java and function names in Haskell are different, so that we need to translate them where incompatible.

The actual naming scheme that is chosen in this paper is the following:

Fields For every public field a function with the name `get'<field>` is created, where `<field>` is the name of the field. Since static and non-static fields share a common namespace in Java we do not have to distinguish them in this regard. For non-final fields also a setter is also created: `set'<field>`.

Methods For every public method a function is created. Since no name in Java can contain the apostrophe they are guaranteed not to clash with getters and setters. Since functions in Haskell are only allowed to begin with a lower case letter, a function that would otherwise start with an upper case letter is prefixed with an underscore (`_`).

Constructors Every constructor is translated into a name of the form `new'<type>`, where `<type>` is the name of the respective class.

Overloading Since we do not intend to overload function names in Haskell, every overloaded name is split up into as many names as the name is overloaded. The names are distinguished by appending a distinct amount of apostrophes. This may lead to funny names like `print''''''` for heavily overloaded methods.

Keywords Whenever a translated name would be equal to a reserved word in the Haskell language, it is prefixed with an underscore (not that there are no upper case keywords in Haskell, so this will not produce any conflicts with upper case method names, which are also augmented with a leading underscore).

General remark Java names are allowed to contain the dollar sign. However, the actual use of the dollar sign in any name is frowned upon and may in fact lead to problems even within the Java ecosystem, since the virtual machine uses the dollar sign to distinguish auto generated methods and classes from user generated ones. It is a *leaky abstraction.* Since there is no method or field name within the Java standard library which actually contains a dollar sign the matter is not very urgent. For completeness however we will replace any occurrence of the dollar sign with two apostrophes. Two apostrophes are chosen since a single apostrophe is already used for distinguishing the namespace of getters, setter, methods, and constructors.
The alert reader may have noticed that there is a slight possibility of a name clash given by this naming scheme: A method `print$` and the third overloaded method with the name `print` could result in the same name. Because of this we will replace a trailing dollar sign not with two apostrophes, but with an apostrophe and an underscore.

3.1.2 The Java Monad

Every action in the Java programming language is executed within the *Java virtual machine* (JVM). Every action may have a side effect that alters the state of the virtual machine. Thus executing a Java action will require the presence of a virtual machine.

A suitable representation for this is to create a custom monad, the Java monad. In the Java monad a reference to a virtual machine is available, it is therefore a State monad (where the state is the state of the virtual machine or a reference to the virtual machine). Since interacting with a virtual machine is an IO action and the virtual machine itself can perform IO actions all the time it also needs to be used within the Haskell IO monad.

We will therefore define the Java monad as a `StateT` monad transformer that wraps the IO monad and has a custom state, a `JVMState`:

```haskell
newtype Java a = Java (StateT JVMState IO a)
    deriving (Monad, MonadState JVMState, Functor, MonadIO)
```

A function `runJava :: Java a -> IO a` can be used to run a computation in the Java virtual machine inside the IO monad.

3.1.3 Primitive Types

In principle every primitive can be mapped one to one to a type of the Haskell 2010 base modules `Data.Int`, `Data.Word`, `Double`, `Float`, or `Bool`.

byte	Data.Int.Int8	boolean	Prelude.Bool
short	Data.Int.Int16	char	Data.Word.Word16
int	Data.Int.Int32	float	Prelude.Float
long	Data.Int.Int64	double	Prelude.Double

While this would be an adequate solution it does not completely capture how Java deals with primitive types. In particular Java has a limited notion of subtyping for primitive types. Also Java has object oriented equivalents for each of the primitive types and automatically converts them back and forth as appropriate.

To capture this behavior we would like to overload primitive types, much in the same way that numeric types are overloaded in standard Haskell via the type classes `Num`, `Frac`, etc. Thus we create a type class for every primitive type and add instances for all Haskell types that can be used like such a type.

Here is how an exemplary `JInt` type class could look like, which resembles the Java native type `int`.

```haskell
class JInt a where
    toInt :: a -> Java Int32 -- java int is a 32 bit signed integer.
```

It is worth noting that – in contrast to e.g. `fromInteger` of the `Num` type class – the function is not a pure function, but a monadic function which executes in the Java monad. This is useful since it will allow us to implement *auto boxing.* Auto Boxing in Java is a feature that allows a primitive type to be automatically converted into an object type. An object however has to be allocated on the heap in the virtual machine, which is a side effect.

It will further allow us to transparently convert between Java data structures and Haskell data structures such as `Map<k,v>` and `[(k,v)]` or `java.lang.String` and `[Char]`.

3.1.4 Objects

Objects may be represented as `newtype` declarations which hold a reference to the actual object (since that is living in the Heap of the virtual machine). In the next sections is, among other things, a mechanism introduced for simulating subtype polymorphism using type classes which involves the creation of a type class for every Java type. Thus every Java type introduced several types in Haskell which we will distinguish by appending an apostrophe to one of them:

```
newtype String' = String' ObjectReference
class String a where ...
```

3.1.5 Subtype Polymorphism

The subtype polymorphism in Java effectively makes every method in Java a polymorphic method. Whenever a parameter of type `X` is declared it may be given every value of a type `X` or a subtype of `X`. What this means is: The type of the value must at least support the same class interface as the type `X`.

A suitable translation from Java to Haskell would therefore be: For each class interface a type class shall be defined that resembles the class interface. Wherever a Java type `X` is mentioned, replace it with a polymorphic type variable that is restricted with a context that requires the type of the variable to have an instance of the corresponding type class.

A suitable translation of the signature for the Java method

```
String getenv(String)
```

would therefore be:

```
getenv :: forall a b (String a, String b) => a -> b
```

It is worth noting that *a* and *b* are distinct type variables here, since it is entirely possible that they do not refer to the same type but two different types which both happen to be a subtype of String (in this particular example this will not be the case since String is declared *final* in the Java standard library).

There is however a problem with this declaration: When the return type of a function call is polymorphic then in Haskell the caller will determine the actual type. But in Java the callee will determine the type of the returned value, and it will do so at runtime. Thus all we know is that we will get a String:

```
getenv :: forall a. String a => a -> String
```

A function `cast :: a -> Maybe b` could be used to check whether a value is actually a value of type *b* and return the casted result when this is the case or `Nothing`.

But there is still a problem with this declaration: Every reference in Java maybe the `null` reference. Thus a proper translation of the above method signature to Haskell would be:

```
getenv :: forall a. String a => a -> Maybe String
```

Yet this declaration is not perfect. The name `String` is used as a type class in the context and as an actual type in the return type. We fix this by augmenting the type `String` with an apostrophe (a valid name in Haskell):

```
getenv :: forall a. String a => a -> Maybe String'
```

How could the definition for the actual type class `String` look like? For now we will define an empty class `String`:

```
class String a
```

In order to use a type as a String it suffices to declare it an instance of the type class `String`. Here are two such instances:

```
instance String [Char]
instance String String'
```

We will defer the discussion on how the implementation of these instances (and the definition of the above type class) should look like to the section about the implementation of the java-bridge library.

Invoking the static `java.lang.System#getEnv(String)` method in Haskell could now look like the following:

```
main = do
    runJava (getenv "HOME") >>= maybe
        (putStrLn "HOME␣environment␣variable␣not␣defined")
        (putStrLn . javaStringToHaskellString)
```

As a matter of fact it might also be possible that `null` should be passed as an argument to an object parameter. For this case we make `Maybe String'` an instance of the class `String`:

```
instance String (Maybe String')
```

The following invocations of getEnv will now all be legal:

```
main = runJava $ do
    str1 <- new'String "a␣string␣object␣in␣java"
    getEnv "HOME"
    getEnv Nothing
    getEnv str1
    getEnv (Just str1)
```

In order to properly model the subtyping relationships of the `String` type, we should also create instances for all its super types:

```
instance Object String'
instance Serializable String'
instance CharSequence String'
instance Comparable String'
```

The inheritance can also be presented in the declaration of the `String` type class:

```
class (Object a, Serializable a, CharSequence a, Comparable a) => String a
```

3.1.6 Static and Non-static Methods

Static methods can be easily translated by creating a function in Haskell with the same number of arguments as the Java function. Non-static methods are bound to an object and need to be augmented with an additional `this` parameter that allows for passing the reference to the bound object to the function. An example can be seen in figure 23.

```
toString :: Object -> String
```

Figure 23: *The Object.toString() method as a Haskell function.*

Since a method can also be invoked on all subtypes of the type it was declared in, we will design that `this` parameter as a polymorphic argument, just like any other object parameter, as seen in section 3.1.5. An example of this can be seen in 24.

```
toString :: Object this => this -> String
```

Figure 24: *A Java method as polymorphic Haskell function.*

3.1.7 Higher Kinded Types and Type Variables

Java does have higher kinded types through *generics*. An example for that is the type `Collection<E>`. Such types can be translated into Haskell types very easily, since the concept of type variables is roughly the same in both languages. Figure 25 shows a rather contrived example for the Java method `V Map<K,V>.put(K key, V value)`.

```
put :: (Map (this key value), Object key, Object value)
    => this key value -> key -> value -> Java value
```

Figure 25: *Translation of a non-static method involving type variables.*

There is a slight cave-at with this approach though. Java does not give up on polymorphism in the presence of type variables. Thus it is perfectly valid to invoke a java method `void AType<A>.aMethod(A a, A b)` with two arguments that only have `A` as a common super type, but are not actually of the same type.

A translation of this behavior is possible by making the subtyping relationship between two objects explicit. Relations between types can be expressed using an extension to the Haskell standard known as *multi parameter type classes* (`MultiParamTypeClasses` in both GHC and Hugs). An example of this can be seen in 26.

```
aMethod :: (AType (this a), InstanceOfAType a1 a, InstanceOfAType a2 a)
       => this a -> a1 -> a2 -> void
```

Figure 26: *The InstanceOfAType relation realizes subtyping.*

We will however refrain from the use of multi parameter type classes, for several reasons. First of all, they are not standard Haskell. Second, multi parameter type classes are mostly useless regarding type inference and type checking. In most situations they require further extensions like *functional dependencies* or *associated types* to give hints to the type inference engine and the type checker. Even with these extensions they will lead to ambiguity which in turn will require the programmer to give lots of type annotations.

Nonetheless we want to preserve the calling semantics of Java in our translation or at least offer a way to reproduce these. This can be achieved by explicitly down casting values where necessary. A function to so is discussed in Section 3.1.10. Calling a function such as the exemplary method `aMethod` from figure 26 might than look like in figure 27.

```
run :: (AType (x a), Object b, Object c) => x a -> b -> c -> Java ()
run x a1 a2
        aMethod x (coerce a1) (coerce a2)
```

Figure 27

It is worth noting that coerce returns `Maybe b`. This is perfectly well since there exist instances for both `Object` and `Maybe Object`, where `Nothing` denotes the `null` reference. Also, since the lower bound is articulated in the context of `run`, we can be sure that only a `Just Object` can and will be returned.

Constraints on type variables The previous example leads directly to another feature of type variables in Java. Since type variables themselves denote existential types we need a way to express this in Haskell. In Java, a lower or an upper bound on a type variable can be specified using the keywords `extends` and `super` in the declaration of a type variable. Such a thing (escpecially an upper bound induced by `super`) is difficult, if not impossible, to translate in standard Haskell. Here again an explicit translation would involve *multi parameter type classes* that make the subtype or supertype relationship explicit. Without resorting to multi parameter type classes we will simply drop those constraints and resort to `coerce` where necessary.

This is by all means a pragmatic decision. Multi parameter type classes would be an excellent tool for expressing the subtyping relationship, but they induce several problems by themselves. Furthermore we would gain very little: `extends` and `super` are a means to introduce subtyping to the otherwise invariant type variables, something which a Haskell programmer is not used to anyway. Furthermore only very little code actually uses bounded type variables (they are not only complicated in Haskell but also in Java). Making type variables always invariant and requiring explicit coercion where needed seems to be the right thing to do$^{\text{TM}}$, with regard to the usability of the resulting translation.

3.1.8 Arrays

Arrays in Java are a special datatype with special support in the syntax of the Java programming language as well as in the instruction set of the Java virtual machine. While in principle arrays are plain collections with a type of kind `* -> *`, they have special (and unsafe) semantics regarding subtyping:

```
Number[] arr = new Integer[] { 1 };
arr[0] = 2.0;
```

Figure 28: *This results in an ArrayStoreException.*

This does not happen with generic types, since – in contrast to arrays – they are *invariant* in the type of the elements and therefore two arrays with differing element types can not possibly be subtypes of each other.

The reason for why arrays work differently than generics is that they existed long before generics were introduced. As we do not want to reproduce this possible source of defects, we simply treat Arrays as an ordinary object type of kind `* -> *`, as if they were a type `Array<E>`.

On a side note: This would be impossible to do in Java since type variables can not be replaced by primitive types, i.e. there can not be a type `Array<int>`. In Haskell there is no such restriction (technically every type in Haskell is boxed and unboxing is an optimization performed by the compiler, thus the annoying difference of boxed vs. unboxed types does not leak into the type system).

Here is an exemplary signature for a Java method that takes an array of `int` (`int[]`) and returns nothing (`void`):

```
arrayMethod :: JInt a => Array a -> Java ()
```

Setting fields While the type of an array itself should be invariant (which is ensured by the above definition), it should still possible to add values of a subtype of the element type to an array. This is ensured due to the modelling of Java subtyping using Haskell type classes, as discussed in the previous sections. Since arrays are treaded as just another datatype, the same properties apply to the translation of array methods as to the translation of generic types in general.

Two special functions are necessary however: Since Java offers a special syntax for accessing arrays we need a `get` and a `set` method for arrays. As usual we resort to fighting our battle using type classes:

3.1.9 Method Return Types and Exceptions

Checked exceptions are one on the most controversial features of Java. In fact, exceptions in general are a highly discussed topic. The go programming language for

```
class JavaArray array where
    get :: array e -> Int32 -> Java e
    set :: array e -> Int32 -> e -> Java ()

instance JavaArray (Array e) where
    -- implementation is discussed in a later section
```

Figure 29: *A typeclass for Java arrays.*

```
objectArrayMethod :: String s => Array (Array s) -> Java ()
```

Figure 30: *A method void objectArrayMethod(java.lang.String[][]).*

example ditched them altogether in favor of multiple return types. In Haskell exceptions exist, but only within monads. Besides that different styles of exception handling are being used and advocated.

Since every interaction with the virtual machine happens within the Java monad we can always resort to `fail`, which will terminate the currently running program and raise an exception in the IO monad.

An alternative way of handling exceptions is to return a value of type `Either e a`, where `e` is the type of the exception and `a` is the type of the expected non-exceptional value. In Java there is also the possibility of `null` being returned from any method that returns an object (which can be considered an exception too). That would be best reflected as the type `Maybe a` in Haskell.

An adequate translation which combines these features would therefore be:

```
currentThread :: Either Exception (Maybe Thread)
```

There is a hidden agenda in the selection of the above example. While it certainly is an adequate translation it is also a very unpleasant one. Although `currentThread` will never return `null` the user is forced to consider this case. Also `currentThread` will never throw an exception, but again the user is forced to consider this case:

```
runJava $ do
    (Right (Just thread)) <- currentThread
    ...
```

Luckily the return type of a function can be polymorphic in Haskell *and* it is determined by the caller, not the callee. We will use this flexibility and offer different implementations using type classes and overloading:

```
class ObjectResult m where
    toObjectResult :: Either JThrowable (Maybe JObject) -> Java m

instance ObjectResult (Maybe a) where
    -- implementation details are discussed later on

instance ObjectResult (Either JThrowable a) where
    ...
```

The signature for `currentThread` now looks like:

```
currentThread :: ObjectResult object => object Thread
```

The following usages are now all legal:

```
runJava $ do
    (Just thread) <- currentThread
    currentThread >>= \result -> case result of
        (Left exc) -> {- handle exception -}
        (Right value) -> {- handle value -}
```

It is worth noting that in standard Haskell it is not possible to simultaneously define `instance ObjectResult a` (for accessing the return type directly) and `instance ObjectResult (Maybe a)` since these instances overlap (`a` subsumes `Maybe a`). It is nevertheless possible to define such instances using the `OverlappingInstances` extension.

It would also be possible to create a type class for every actual return type, such as `class StringResult` or `class ThreadResult`, which would than allow for a usage like this:

```
runJava $ do
    t <- currentThread :: Java Thread
    t <- currentThread :: Java (Maybe Thread)
    t <- currentThread :: Java (Either (Maybe Thread))
```

3.1.10 A Note on Return Types and Type Casts

The return type of a Java method is actually an existential type, which is not supported by standard Haskell. Therefore return types are not translated in the same way as method parameters, i.e. not polymorphic. Whether a value actually does have a specific type needs to be checked at runtime.

A safe function for checking this and if necessary coercing the value would have a signature like this:

```
coerce :: a -> Maybe b
```

In principle such a function could only ever return `Nothing`, since otherwise it would know how to create a `String` from an `Int`, as well as a `FlyingCarpet` from a `BufferedStringBuilder`. It surely needs some more information to fulfil its duty. We know two things: `coerce` performs a cast on Java objects, and it asks the virtual machine for whether it can perform a certain cast or not. Thus we create a type class for Java Objects which can, for each type, query the virtual machine for whether a given object can be cast to its object type or not:

```
class JavaObject b where
    coerce :: JavaObject a => a -> Java (Maybe b)
```

Every Java type needs to have an instance of `JavaObject` for this to work.

3.1.11 Anonymous Classes

To enable a Haskell programmer to completely make use of a Java library it is crucial that dynamic instances of Java interfaces can be created. This is necessary for example to provide callbacks, for example for event handlers in Swing or for spawning a thread in Java (using the interface `java.lang.Runnable`).

Up until now we only translated functions so that we could call them from Haskell, action would take place in a virtual machine instance. This time we want to create a data object in Haskell which can be exposed to the virtual machine and act like a Java object.

We will not discuss the technical details in this place (we will do so in a later section dealing with the actual implementation of the java-bridge library), but outline the approach.

The simplest case are interfaces which specify exactly one method. A Haskell function itself can be regarded as an instance of such an interface.

Interfaces which specify more than one method need to be modeled as a more complex datatype than one ordinary function. Thus an algebraic datatype for an should be created for interfaces with more than one public method. The names for the method functions are chosen according to the same rules as otherwise.

In order to use these declarations as Java objects, they need to be made an instance of the correspondent type classes. For arbitrary functions this will require the *flexible*

```haskell
myThread :: Java ()
myThread = do
    Thread.sleep (500 :: Int)
    return ()
```

Figure 31: *A perfectly valid Haskell representation of an object that implements java.lang.Runnable.*

```haskell
data Iterable e = Iterable {
    hasNext :: Java Bool,
    next :: Java (Maybe e),
    remove :: Java ()
 }
```

Figure 32: *Representation of a more complex interface: java.lang.Iterable*

```haskell
myIterable :: Iterable Int64
myIterable = Iterable {
    hasNext = do
        return False,
    next = do
        return Nothing,
    remove = return ()
 }
```

Figure 33: *An implementation of the empty iterator.*

instances language Extension of Haskell. If we want to circumvent this, the function has to be wrapped in a `newtype`:

Example with FlexibleInstances:

```haskell
instance Runnable (Java ()) where
    ... -- implementation is discussed later on
```

Using a `newtype`:

```haskell
newtype RunnableFunc = RunnableFunc (Java ())
instance Runnable RunnableFunc where
    ...
```

It is now possible to use such a definition transparently to create anonymous class instances:

```haskell
(Just t) <- new'Thread (RunnableFunc (sleep (500 :: Int32) >> return ()))
start t
```

`FlexibleInstances` make it just slightly more convenient to use:

```haskell
(Just t) <- new'Thread (sleep (500 :: Int32) >> return ())
start t
```

Up until now we only translated Java methods to Haskell functions so that we could call them from within Haskell. In this setting however, the virtual machine is going to call back into Haskell and code written in Haskell is going to be executed. Because of this function signatures will look different, i.e. the function arguments need not be polymorphic. This is due to the fact that inside of the function we will need to access the arguments and we have no clue there what exact type the virtual machine will have given us. For all we know we get types as specified in the signature of a method (this is the same in Java). Accessing arguments as concrete subtypes requires converting them using `coerce`, just as you need to check a value using `instanceof` in Java and doing an explicit cast.

```java
public interface Example<E> {
    List<E> getSequence(int howMuch);
    int count(Collection<E> coll);
}
```

```haskell
data Example e = Example {
    getData :: Int32 -> Java (List e),
    count :: Collection e -> Java Int32
 }
```

Figure 34: *An example interface and its record which can be used as a callback.*

3.1.12 Summary

Up until now we have discussed a scheme for translating Java idioms into Haskell equivalents in order to invoke methods in a virtual machine. We left out so far the actual implementation of this. It will be discussed in section 4 and 5 in which a general library for accessing a virtual machine via the JNI is built and a tool for the automatic generation of bindings to the Java standard library presented (a tool which automatically creates the translation discussed above).

The translation scheme proposed so far compromises the following rules:

- **Fields**, **methods**, and **constructors** are translated into functions. The naming rules from section 3.1.1 make sure that the names do not clash with each other and do not accidentally result in a reserved name of the Haskell programming language.

- Since every action happens inside the Java virtual machine, every translated function is bonded to a special monad, the **Java monad**. The Java monad is a monad transformer on top of the IO monad, since it is strictly more powerful than the IO monad (in addition to IO actions in can perform Java action). This was discussed in section 3.1.2.

- Type classes are to be created for the **primitive types** in order to allow automatic conversion between Haskell values and Java values. Furthermore this approach allows for auto-boxing conventions to be implemented just like in the Java programming language (i.e. a method that takes the primitive type `int` as its first argument can equally well be invoked with an object of type `java.lang.Integer`). The standard representation of primitive types are the basic types from Haskell2010 (Int32, Char, etc.). This was discussed in section 3.1.3.
  It is worth noting that it may seem rather irritating that the type classes being defined do not contain any functions at all. It would be completely possible to add the non-static methods of an interface or a class to the type class definition. This does however not include the non-static interfaces. Furthermore we will see later on that we run into trouble implementing such type classes due to mutually recursive definitions. The sole purpose of type classes and instances is therefore to model the inheritance hierarchy.

- **Subtyping** is simulated using Haskells type classes. For each Java type, a Haskell type class is to be created. A concrete object type is made an instance of all type classes that resemble supertypes or interfaces implemented by that type. This was discussed in section 3.1.5.

- **Generic types** are more-or-less translated as is, since type variables are a common feature of Java and Haskell. Upper and lower bounds on type variables are discarded in order to reduce the complexity of the resulting translation. To compensate for this a coercion function `coerce` has to used which is defined in a type class `JavaObject` of which every object type is a member. This was discussed in section 3.1.7.

- While **arrays** are a specialized datatype in its own right in the Java programming language, they are treated as an ordinary generic type, parameterized in its component type. This was discussed in section 3.1.8.
- Polymorphic return types are exploited to let the programmer decide how to handle **exceptions**. For each type a type class `<Type>Result` is created (in case of objects the clutter can be reduced to a single `ObjectResult` class). If an exception is discarded by the programmer, an exception is raised using the native monadic `fail` function. This mechanism will also allow convenient marshalling between Haskell values and Java objects. This was discussed in section 3.1.9.
- **Callbacks** that allow the virtual machine to call back into the enclosing Haskell program are created by declaring functions or records as an instance of the corresponding type class that resembles the interface to be implemented by these functions. This was discussed in section 4.4.5.
- Sometimes it is necessary to perform **casts** between types. This situation arises in both Java as in Haskell. A type safe `coerce` function is proposed which checks whether a cast is possible and performs it if that is so.

```
newtype Object' = Object' ObjectReference
class Object a
instance Object Object'

toString :: (Object this, ObjectResult (object String'))
    => this -> object
equals :: (Object this, Object a1, BooleanResult boolean)
    => this -> a1 -> boolean
-- ... further object functions ...

newtype String' = String' ObjectReference
class (Object a) => String a
instance Object String'
instance String String'
-- ... string method functions ...

newtype Map' k v = Map' ObjectReference
class Map' a
instance Map (Map' k v)

entrySet :: (Map (this k v), ObjectResult (object (Set' (MapEntry' k v))))
    => this k v -> object (Set' (MapEntry' k v))
-- ... further map method functions ...
```

Figure 35: *A (shortened) exemplary translation of certain Java types.*

3.2 Translating Haskell idioms into Java

The public API of a Haskell package consists of a set of *modules*. A module declares a public interface, that is a set of public functions and types.

Translating such an interface into Java requires finding a suitable representation of Haskell idioms in the Java language.

3.2.1 There is only Data

In Haskell there is only data which may be of a certain type. That type in turn has a certain kind, like `*`, or `* -> * -> *`. The kind of a type simply specifies how many arguments a type constructor takes. We translate that into Java by creating a Java type for each Haskell type and adding as many type variables as necessary to capture the kind of that type.

Certain datatypes – functions, lists, and tuples lists – have a special syntax in Haskell (`->`, `[]`, `(,,)`). They will be translated using default names like `Function`, `List`, or `Tuple3`.

Here are some example translations:

Haskell	**Java**
`Map k v`	`Map<k,v>`
`Int32`	`Int32`
`Either (Maybe a) [String]`	`Either<Maybe<a>,List<String>>`
`(a, b, c)`	`Tuple3<a,b,c>`
`a -> (Int,[b])`	`Function<a,Pair<Int,List<b>>>`
`a -> a -> a`	`Function<a,Function<a,a>>`

There is a slight deficiency of this translation scheme: Java does *not* support higher kinded type variables, i.e. a function like `(a -> b) -> f a -> f b` can not be translated as `Function<Function<a,b>,Function<f<a>,f<b>>>`, since `f` is a type variable and can not be parameterized.

This can be partially remedied, see section 3.2.3 for that.

3.2.2 Algebraic Datatypes and Constructors

Algebraic datatypes are the bread and butter of Haskell. They are constructed using constructors. A constructor in Haskell is similar to a constructor in Java, yet completely different. A constructor in Java is associated with the initialization of the state of an object, whereas a constructor in Haskell is simply a function that creates objects (and since there is no state in Haskell, no initialization is done).

```
data ParseResult a = Error  { line: Int, message: [Char] }
                   | Result { info: Info, tokens: a }
```

Figure 36: *An algebraic datatype in Haskell.*

Algebraic datatypes are also known as *discriminated unions* or *tagged unions*. This basically means that one type can store different types and it knows which type it actually stores.

Figure 37 shows how such a tagged union could look like in the C programming language.

While this declares how the data is represented in memory (something we do not have to worry about using Haskell), this is not everything that is being declared in figure 36. In fact, there are actually **seven** things being declared here:

1. A type constructor `ParseResult :: * -> *`

```c
enum parse_result_e { ERROR, RESULT }
union ParseResult {
    enum parse_result_e type;
    struct {
        line_t line;
        char* message;
    };
    struct {
        into_t info;
        result_t tokens;
    };
}
```

Figure 37: *A tagged union in C.*

2. A constructor function `Error :: Int -> [Char] -> ParseResult a`
3. A constructor function `Result :: Info -> [a] -> ParseResult a`
4. An accessor function `line :: ParseResult a -> Int`
5. An accessor function `message :: ParseResult a -> [Char]`
6. An accessor function `info :: ParseResult a -> Info`
7. An accessor function `tokens :: ParseResult a -> [a]`

For a complete translation to Java each of these things has to be translated:

1. The type itself can be translated into an abstract class with a private constructor. The constructor has to be private in order to prevent creation of subclasses outside of the definition, since algebraic datatypes can not be extended after their declaration.
2. Each constructor yields an enclosed class and a static factory function. The constructors of the enclosed classes again are private, in order to prevent instantiation of these classes in other ways than by the factory functions. By declaring classes for every constructor, simple pattern matching can be simulated by means of the `instanceof` operator in Haskell.
3. For each field that a record can hold, a private final field is generated in the corresponding member class. A public getter is defined for each field in both the abstract class (marked as abstract) as in the member class representing the corresponding Haskell constructor.
4. The getter functions throw an error if invoked on a member class which actually does not have the corresponding field (this is the same behavior as in Haskell).

The algebraic datatype `Maybe` can now be translated as can be seen in figure **??**.

3.2.3 Type Classes

Type classes can be understood as interfaces. A Haskell type can be part of multiple type classes, and so a Java type can implement multiple interfaces. Furthermore there is a problem with higher kinded type variables – there are non in Java. It turns out however, that higher kinded type variables almost never make any sense if not constrained by a certain context, i.e. such type variables are always part of a certain type class (or multiple type classes).

The monadic return function `a -> m a` for example can not return any type at all without knowing how to create `m` values. It is hence part of the type class `Monad` which provides this information (if there is an instance for the desired type).

```
data Maybe a = Nothing | Just a
```

```
public abstract class Maybe<A> {
    private Maybe() {}
    public static final class Nothing<A> extends Maybe<A> {
        private Nothing() {}
    }
    public static final class Just<A> extends Maybe<A> {
        private final A a1;
        private Just(A a1) {
            this.a1 = a1;
        }
    }
    public static <A> Nothing<A> Nothing() {
        return new Nothing<A>();
    }
    public static <A> Just<A> Just(A a1) {
        return new Just<A>(a1);
    }
}
```

Figure 38: *Translation of the algebraic datatype Maybe to Java.*

If type classes are translated using Java interfaces our problem with higher kinded type variables goes away.

Consider the type class Functor:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

It can be translated to the following Java interface:

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

This definition however needs to be relaxed a bit. Since parameterized types are invariant in their type parameters in Java we can not create an appropriate implementation of Functor for e.g. a `List<A>` class. The type parameters need to be made covariant, i.e. allow for a subtype in order to properly implement `fmap`.

```
public interface Functor<A> {
    public <B> Function<? extends Functor<A>,? extends Functor<B>>
        fmap(Function<A,B> f);
}

public class List<A> implements Functor<A> {
    public <B> Function<List<A>, List<B>> fmap(final Function<A,B> f) {
        return ...
    }

    /* remaining methods */
}
```

Figure 39: *The Haskell type class "Functor" as an interface in Java.*

Figure 39 shows a proper example of the `Functor` interface, including an (incomplete) implementation of a fictitious `List` class (it concentrates on the types only here):

This way we not only have resolved the issue with higher kinded type variables, but also we eliminated the context `Functor f =>` by moving it into the signature.

There is a cave-at though. Consider the following, admittedly rather complex, function signature taken from the Parsec parser combinator library:

```
runParsecT :: Monad m => ParsecT s u m a -> State s u -> m (Consumed (m (Reply s u a)))
```

The problem in this definition is that this time ”m” can not be replaced like in the previous example. In fact, there really is no higher kinded type variable in the definition of `fmap`. The type `f a` has the ordinary kind `*`, which is why we could replace it with a Java type of the same kind (`Functor<A>`). But `m` in the signature of `runParsecT` really *has* the higher kind `* -> *`.

Furthermore: If we replaced `m` in the signature of `runParsecT` where it is actually applied to a type argument, we would lose the ability to express that the first `m` in the return type is exactly the same type as the second m (again of kind `* -> *`).

Bacause of this shortcoming in Java generics certain (admittedly contrived) types can not be properly resembled in Java. Most prominently it is not possible to properly express monad transformers (they always take a type constructor somewhere, and in fact, `runParsecT` was not chosen without second thoughts, since `ParsecT` is in fact a monad transformer that wraps any given monad `m`).

3.2.4 Polymorphic Return Types

Except for type constructors and higher kinded type variables, the translation of a Haskell interface into Java seems to be rather straight forward till now. There is however one further corner case which does not have a translation as obvious as the exemplary types from the beginning of this section: Polymorphic return types. Consider the following two examples:

```
id :: a -> a
minBound :: Enum e => e
```

Translating `id` indeed *is* straight forward:

```
public static <A> A id(A a);
```

`minBound` howevers poses a problem:

```
public static <E> E minBound();
```

What exactly should `E` be? In Java it is not possible to select the type like in Haskell via `minBound :: Int`. There are only three things that can be returned here: (1) a value of type Object, (2) the null reference, (3) throw an exception.

This is very similar to a Haskell function `f :: Maybe e`. This function can only return (1) `Nothing`, (2) or $\perp$ (In the Java case the additional third value of type `Object` is in the set of possible return values since generic types are always at least an object, however a plain Object is just about as meaningless as the null reference).

But: We missed something! In contrast to `f`, `minBound` provides some additional info: `e` should be a member of the type class `Enum`. Lets reconsider our translation to Java and include that extra information:

```
public static <E> E minBound(Class<E>);
```

The java compiler can now deduce that the return type of `minBound` must be the same type as the type parameter to `Class`. Also it is now possible to pick a value for the return type, though it has to be done at runtime, whereas the selection of an appropriate instance can be done at compile time in Haskell.

3.2.5 Summary

While a possible translation of Haskell to Java seems feasible, even easy, at first, it turns out it is not. Although the core of Haskell is a very simple language (we have seen that everything is data of a certain type, types have certain kinds, and that's pretty much it), the devil is in the details. Expressing type constructors and higher kinded type variables in Java is simply not possible (or at least not in all cases).

We have not yet looked at

- how to translate names from Haskell to Java;
- how to deal with the different execution models of Haskell and Java;
- how to deal with exceptions.

Since both Haskell and Java do have exception, we could simply raise a `HaskellException` if anything goes wrong when invoking a Haskell function.

The different execution models are a bit tougher: While in principle very similar, there are corner cases which may go wrong. As we have seen, Haskell can have infinite lists and make something useful out of a computation that would not return in a strict execution model. Care must be taken when translating such a function, e.g. by translating it not as a list but as an `Iterable`. The most important thing to keep in mind is that the invocation should not force the value to be evaluated as this would be exactly what could result in an infinite loop. Instead we have to reproduce the semantics by implementing explicit laziness.

The most difficult part is the translation of names from Haskell to Java. Haskell allows the programmer to define arbitrary operators, and the authors of many Haskell libraries do use this feature exhaustively. In order to translate such functions we would have to come up with a translation scheme that in any case will look ugly to the eye of both the trained Haskell as well the fine Java programmer.

4 The java-bridge library

Auch aus Steinen die in den Weg gelegt werden kann man Schönes bauen
– Johann Wolfang v. Goethe

The `java-bridge` library is a Haskell library that allows to retrieve references to classes and objects in the Java virtual machine and invoke methods on objects or classes. It also supports passing function pointers from the Haskell runtime to the virtual machine in order to allow Java code to call back into Haskell. Most importantly the library ensures that garbage collection still works correctly across runtime boundaries.

4.1 Prerequisites

As a matter of fact the virtual machine is a highly concurrent application, since Java has been designed from the ground up with concurrency in mind. The Haskell compiler being used will thus have to have basic support for concurrency, i.e. it will have to emit thread-safe programs which at least do not crash when the garbage collection mechanism is invoked concurrently from the outside.

Unfortunately the only Haskell compiler that currently supports concurrency is the Glasgow Haskell Compilation System (GHC). The implementation of the java-bridge is hence restricted to GHC. Most Haskell applications are GHC-centric these days anyways which is also due to the huge amount of language extensions that GHC supports. Many libraries and modern Haskell idioms (such as monad transformers) depend on extensions such as *multi parameter type classes* or *functional dependencies* and it is expected that these extensions are being incorporated into the language standard soon. In fact, Haskell was designed as a testbed for experimentation with those language extensions and it is commonly seen among Haskell compilers that proprietary extensions are being implemented. The ongoing effort to integrate these extensions into the language standard is known as *Haskell Prime (Haskell')* and has already resulted in a minor revision of the Haskell language, the Haskell 2010 language standard. Haskell 2010 incorporates the *foreign function interface* extension, which is another necessary prerequisite for the java-bridge library. A major revision of Haskell is expected next year, Haskell 2014.

4.1.1 The Haskell Foreign Function Interface

The Haskell FFI describes some syntactic additions to the Haskell language that allow the programmer to import definitions from other languages and mark Haskell functions as exported, i.e. callable from the outside world. The FFI in its current form only describes the interfacing with programs written in the C programming language. In comparison to the JNI it is strictly less powerful, as it does not allow for an arbitrary function to be called from the outside world but only for a function that is explicitly marked as such.

It can be assumed that the primary motivation of the creators of the FFI was to allow for Haskell applications to interface with native libraries in order to enrich the Haskell programming experience. On the other hand there really are immense technical hurdles to overcome would one really want to employ a mechanism that allows for calling arbitrary functions.

The situation is not that worse though, the FFI is absolutely capable of turning arbitrary functions into `FunPtr` values, as well as it can reference any value by a `StablePtr`. This can however only work from within Haskell, i.e. it is not possible to query a module from the outside for its functions and invoke one of them.

4.1.2 The Java Native Interface

The JNI comprises an API for accessing the JVM and for accessing native methods from within the JVM. We will need to use the JNI in order to invoke functions within the virtual machine form the outside world as well as to call native methods from within the virtual machine (the JNI allows for both). The JNI offers a very rich interface and, most importantly, offers access to literally everything inside the virtual machine. Paired with the JVMs powerful reflection methods it is possible to examine every bit of the JVM.

4.2 Design of the Library

Since both Haskell and Java only support interfacing with programs written in the C programming language, the most basic parts of the java-bridge library are to be written in C.

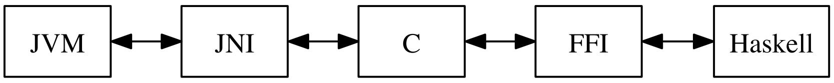


Figure 40: *Basic data flow between Haskell and Java*

This will of course require some glue code in both Haskell and Java to be written:

A low level binding to the JNI will be given in Haskell which is referred to as the *low level interface*. On top of the low level interface a *medium level interface* is to be built which effectively comprises a simple DSL (*domain specific language*) for method lookup and invocation. The medium level interface will also hide the details of reference management and garbage collection from the enduser of the library.

On the Java side of things there will be some classes for abstracting basic Haskell datatypes (the function, basically).

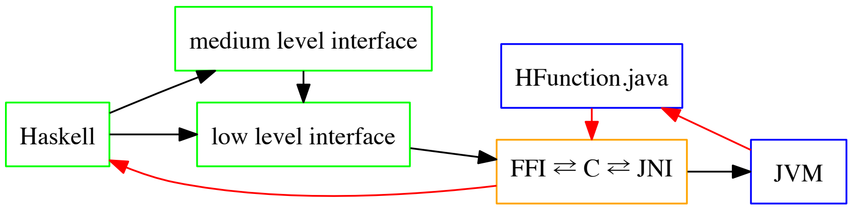


Figure 41: *The interaction of components in the java-bridge library. Green components are written in Haskell, yellow ones in C, blue ones in Java. Black arrows are calls from Haskell into the JVM, read arrows are calls from the JVM back into Haskell.*

4.3 Low Level Interface

The core of the java-bridge library is a direct translation of the JNI into Haskell. These bindings are located in the modules `Foreign.Java.JNI.Safe` and `Foreign.Java.JNI.Unsafe`. This subsection comprises a short walk through the API defined in these two modules.

4.3.1 IDs and References

Every function in the low level interface requires a pointer to an instance of the JVM. The Haskell FFI represents pointers as the type `Ptr a`, where a is the type of the data the pointer is pointing at (i.e. pointers are to some extent type safe in Haskell).

Haskell 2010 allows for *phantom types*, i.e. types that are not actually inhabited by any non-bottom value. A phantom type is introduced by an empty data declaration. Such types can be used to parameterize pointers adequately. Throughout the low level interface the following phantom types are used:

JVM
An instance of the Java Virtual Machine.

JConstructorID
The id of a Java constructor.

JFieldID, JStaticFieldID
The id of a Java field.

JMethodID, JStaticMethodID
The id of a Java method.

JObjectRef, JClassRef, JThrowableRef
A reference to an object, a class, or an exception.

The JNI distinguishes several reference types: (1) *local references*, (2) *global references*, (3) *IDs.*

IDs are stable pointers that will not ever be garbage collected and need not be released manually.

References reference an object in the heap space of the JVM (well, actually classes are allocated in the permanent generation space of the Java virtual machine, a part of memory that is much more seldom visited by the garbage collector, but we can view this as a special part of the heap). References are not ordinary pointers, but pointers to pointers in the JVM. This is necessary so that the JVM can do some book keeping and knows which objects it may not yet garbage collect since they are probably still references by some foreign code (otherwise the garbage collector might collect objects which are not referenced by any other object in the Java heap anymore and we would end up with a dangling pointer).

Local references are references that are automatically destroyed after the function they were created in returns. Implementations of the JNI achieve this by allocating a local reference table that is destroyed as soon as the calling function returns. These references are not intended to have a long lifetime and in fact are invalid as soon as they are returned by a function (much like returning a pointer to locally allocated data). Obviously local references are unsuitable for use within Haskell, since once they end up in the Haskell runtime they have surely left the realm of the C function they were created in.

Global references on the other hand are stable references and will live as long as they are not manually released. These are the references that we are going to deal with, and therefore we will have to free these resources by hand. All functions in the low level interface return global references. Since most functions in the JNI return local references the C glue code converts them into global references.

4.3.2 Controlling the Java Virtual Machine

Before we can do anything with a virtual machine, we need to create one. Afterwards we need to tear it down again. This is what the functions `createVM`, `createVM'`, and `destroyVM` are for.

```
function jobject localToGlobal(vmi_t* vmi, jobject local) {
    jobject global = NULL;
    if (local == NULL) {
        global = NULL;
    } else {
        global = (*vmi->env)->NewGlobalRef(vmi->env, local);
        (*vmi->env)->DeleteLocalRef(vmi->env, local);
    }
    return global;
}
```

Figure 42: *C code that converts a local reference to a global reference. This is actually not a function in the java bridge library but a macro vmi.* `vmi_t` *is the virtual machine interface that is maintained by the library.*

createVM `IO (Ptr JVM)`
Creates a new virtual machine instance. Note that in theory it should be possible to create multiple virtual machines within a process, in practice no known implementation of a virtual machine does support this; thus this function should be called at most once.

createVM' `Word32 -> Ptr CString -> IO (Ptr JVM)`
Like `createVM` but additionally takes an array (`Ptr CString` = `char**`) of arguments. The length of this array has to be specified by the first parameter (just like in C since this is a very low level binding).

destroyVM `Ptr JVM -> IO ()`
Destroys a virtual machine instance.

persistVM `Ptr JVM -> IO ()`
This function is not present in the JNI. It is a special function that is needed by the medium level interface to tell the C parts of the java-bridge that the virtual machine should not be teared down after the last environment pointer is released. The JNI differentiates between pointers to the virtual machine and execution environments. Each environment pointer is attached to a specific thread and vice versa. In order to allow the concurrent access from multiple Haskell threads to the virtual machine multiple environment pointers have to be maintained by the c library which will automatically destroy the VM if the last environment pointer is released.

4.3.3 Class, Method and Field Lookup

findClassRef `Ptr JVM -> CString -> IO (Ptr JClassRef)`
Discovers a class. The method requires a *binary name*[23] as the name of the class to find, but with slashes instead of dots i.e. `java/lang/Thread$State`. If the class could not be found this function will return a null pointer (`nullPtr` in Haskell).

getConstructorID `Ptr JVM -> Ptr JClassRef -> CString -> IO (Ptr JConstructorID)` Find the *constructor id* based on a method descriptor. A method descriptor is an encoding of the signature of a method like it is stored inside a Java class file. For example the method descriptor for the main function of a Java program is `([Ljava.lang.String)V` (a method `(...)` that takes an array `[` of objects `L` of type `java.lang.String` as argument and returns nothing, i.e. void `V`). If the constructor does not exist a null pointer is returned.

getStaticMethodID `Ptr JVM -> Ptr JClassRef -> CString -> CString -> IO (Ptr JStaticMethodID)`
Find the *method id* of a static method. The first `CString` argument is the method descriptor, the second is the name of the method. Returns the null pointer if the method does not exist.

getMethodID `Ptr JVM -> Ptr JClassRef -> CString -> CString -> IO (Ptr JMethodID)` Find the *method id* of a non-static method. The first `CString` argument is the method descriptor, the second is the name of the method. Returns the null pointer if the method does not exist.

getFieldID `Ptr JVM -> Ptr JClassRef -> CString -> CString -> IO (Ptr JFieldID)` Find the *field id* of a static method. The first `CString` argument is the field descriptor, the second is the name of the field. Returns the null pointer if the field does not exist.

[23]Java Language Specification §13.1 The Form of a Binary

getStaticFieldID `Ptr JVM -> Ptr JClassRef -> CString -> CString -> IO (Ptr JStaticFieldID)`
Find the *field id* of a static field. The first `CString` argument is the field descriptor, the second is the name of the field. Returns the null pointer if the field does not exist.

4.3.4 Invoking Methods

Calling methods comprises 22 functions, 11 for each of the basic types, times two for static and non-static methods. Each function takes a pointer to the JVM, a class or object reference, and an array of values. The different types in the name are designated by `<T>` and the result is designated by `<R>`.

callStatic<T>Method `Ptr JVM -> Ptr JClassRef -> Ptr JStaticMethodID -> Ptr JValues -> IO <R>` There are actually 11 functions, one for each primitive type, one for `void`, one for `Object` and one for `String` (the low level interface has some special interface for dealing with strings). Invoke a static method. A static method is invoked on a class, thus you have to give it a pointer to a class reference. The `JValues` argument is an array of values that are passed to the function. See *argument passing (4.3.6)* for further info. Returns the result of the method call (of type `<R>`, i.e. one of Int8, Int16, Int32, Int64, Float, Double, Word16 (a java char), Bool, a `Ptr JObjectRef` (which might be the `nullPtr`) or a `CString` (actually a `Ptr Char`)).

call<T>Method `Ptr JVM -> Ptr JObjectRef -> Ptr JMethodID -> Ptr JValues -> IO <R>` There are actually 11 functions, one for each primitive type, one for `void`, one for `Object` and one for `String` (the low level interface has some special interface for dealing with strings). Invoke a static method. Invoke a static method. A static method is invoked on an object, thus you have to give it a pointer to an object reference. The `JValues` argument is an array of values that are passed to the function. See *argument passing (4.3.6)* for further info. Returns the result of the method call (of type `<R>`, i.e. one of Int8, Int16, Int32, Int64, Float, Double, Word16 (a java char), Bool, a `Ptr JObjectRef` (which might be the `nullPtr`) or a `CString` (actually a `Ptr Char`)).

4.3.5 Getting and Setting Fields

Reading and writing fields actually comprises 44 functions, 11 for the different basic types (including void and String, as there is some special support for Strings), times two for static and non-static fields, times two for reading or writing. Each of these functions takes a pointer to the JVM, class reference for static fields or an object reference for non-static fields, and the (if it is a setter) the value to be set. The different types in the name are denoted by `<T>`, in the result type is given as `<R>`.

get<T>Field `Ptr JVM -> Ptr JClassRef -> Ptr JStaticFieldID -> IO <R>`
There are actually 10 functions, one for each primitive type, one for `Object` and one for `String` (the low level interface has some special interface for dealing with strings). Reads the value of a static field. Returns that value of type `<R>`, i.e. one of Int8, Int16, Int32, Int64, Float, Double, Word16 (a java char), Bool, a `Ptr JObjectRef` (which might be the `nullPtr`) or a `CString` (actually a `Ptr Char`).

set<T>Field `Ptr JVM -> Ptr JClassRef -> Ptr JStaticFieldID -> <A> -> IO ()`
There are actually 10 functions, one for each primitive type, one for `Object` and one for `String` (the low level interface has some special interface for dealing with strings). Sets a non-static field to a new value. Returns nothing, i.e. `()`.

getStatic<T>Field `Ptr JVM -> Ptr JClassRef -> Ptr JStaticFieldID -> IO <R>`
There are actually 10 functions, one for each primitive type, one for `Object` and one for `String` (the low level interface has some special interface for dealing with strings). Reads the value of a static field. Returns the result of that field of type `<R>`, i.e. one of Int8, Int16, Int32, Int64, Float, Double, Word16 (a java char), Bool, a `Ptr JObjectRef` (which might be the `nullPtr`) or a `CString` (actually a `Ptr Char`).

setStatic<T>Field `Ptr JVM -> Ptr JClassRef -> Ptr JStaticFieldID -> <A> -> IO ()`
There are actually 10 functions, one for each primitive type, one for `Object` and one for `String` (the low level interface has some special interface for dealing with strings). Sets a non-static field to a new value. Returns nothing, i.e. `()`.

4.3.6 Argument Passing

Every function that passes multiple arguments to the JNI requires a `JValues` pointer, i.e. an array of multiple values. Such arrays are represented as lists of `JArg` values in Haskell, where `JArg` is given by the following algebraic data type:

```
data JArg
  = BooleanA Bool
  | CharA    Word16
  | ByteA    Int8
  | ShortA   Int16
  | IntA     Int32
  | LongA    Int64
  | FloatA   Float
  | DoubleA  Double
  | StringA  String
  | ObjectA  (Maybe JObject)
```

mkJValues `Ptr JVM -> [JArg] -> IO (Ptr JValues)`
Creates a new JValues pointer from the list of JArgs.

setJValue<T> `Ptr JValues -> CInt -> <A> -> IO ()`
These are actually 9 functions, one for each type as distinguished by the datatype `JArg`, except for string. The `CInt` argument takes the index of the which array element is to be updated. `<A>` denotes the actual type to be set.

setJValueString `Ptr JVM -> Ptr JValues -> CInt -> CString -> IO ()`
The low level JNI interface has some special suppport for Strings. This is purely for convenience. Creating a `String` object in the virtual machine requires an instance of the virtual machine, thus this function takes an additional pointer to the JVM as its first argument.

4.3.7 Releasing Resources

Since the low level interface returns global references only, we needs to release these references manually. Since we introduced some basic type safety (by distinguishing at least `JObjectRef`, `JClassRef`, and `JThrowableRef`) we need to create one function for each of this types:

releaseJObjectRef `Ptr JVM -> Ptr JObjectRef -> IO ()`

releaseJClassRef `Ptr JVM -> Ptr JClassRef -> IO ()`

releaseJThrowableRef `Ptr JVM -> Ptr JThrowableRef -> IO ()`

release `FunPtr (Ptr a -> IO ())`
This is a special function which is needed by the medium level interface in order to automatically release references. Haskell features a special type `ForeignPtr` which can be associated with a `FunPtr` which is invoked by the Haskell garbage collector when a `ForeignPtr` is no longer in use in the Haskell runtime. A pointer to the following C function is returned:

```
void release (jobject obj)
{
    JavaVM* jvm;
    jsize numCreatedVMs = 0;

    GET_CREATED_JAVA_VMS(&jvm, 1, &numCreatedVMs);
    if (numCreatedVMs == 1) {
        JNIEnv* env;
        if ((*jvm)->GetEnv(jvm, (void**) &env, FFIJNI_VERSION) == JNI_OK) {
            (*env)->DeleteGlobalRef(env, obj);
        } else {
            // The thread is no longer attached.
            // This can only happen if the virtual machine
            // is being shut down by ‘destroyVM‘ but has not
            // yet been shut down completely.
        }
    } else {
        // The virtual machine has already been shut down.
    }
}
```

4.3.8 Support for Complex Datatypes

While the JNI does have no special treatment for the String data type, the low level interface does offer `callStaticMethod`, `getField`, etc. for the String type, too. Arrays on the other hand are handled as ordinary objects. It is however possible to treat Strings just like `JObjectRef`s, thus there are some functions to assist the programmer with these objects.

newJString	`Ptr JVM -> CString -> IO (Ptr JObjectRef)`
charsFromJString	`Ptr JVM -> Ptr JObjectRef -> IO (Ptr JChars)`
bytesFromJString	`Ptr JVM -> Ptr JObjectRef -> IO (Ptr JBytes)`
releaseJChars	`Ptr JVM -> Ptr JObjectRef -> CString -> IO ()`
releaseJBytes	`Ptr JVM -> Ptr JObjectRef -> CString -> IO ()`
jstringToCString	`Ptr JVM -> Ptr JObjectRef -> CString -> IO ()`
new<T>Array	`Ptr JVM -> Int32 -> IO ()` The `Int32` parameter takes the length of the array.
newObjectArray	`Ptr JVM -> Int32 -> Ptr JClass -> Ptr JObjectRef` Additionally takes a pointer to a class in order to declare an array for a specific component type (e.g. `String[]`). The array elements are initialized with `null`.

4.3.9 Reflection

These methods allow for retrieving information about Java objects.

getObjectClass	`Ptr JVM -> JObjectRef -> IO (Ptr JClassRef)`
isInstanceOf	`Ptr JVM -> Ptr JObjectRef -> Ptr JClassRef -> IO Bool` Checks whether the object referenced by `JObjectRef` is an instance of the class `JClassRef`.

4.3.10 Exception Handling

These functions are literally translated from the JNI, except for `exceptionOccurredClear`. Typically one has to check manually for whether an exception has occurred or not (via `exceptionCheck`), than retrieve the exception (via `exceptionOccurred`) and clear the exception in the JVM (via `exceptionClear`). `exceptionOccurredClear` replaces this three-step procedure with a single one that returns a `nullPtr` if no exception occurred, and has the side effect of clearing the exception state if it did not return null.

exceptionCheck	`Ptr JVM -> IO Bool`
exceptionOccurredClear	`Ptr JVM -> IO (Ptr JThrowableRef)`
exceptionClear	`Ptr JVM -> IO ()`
exceptionOccurred	`Ptr JVM -> IO (Ptr JThrowableRef)`

4.3.11 Callbacks

The C part of the java-bridge library offers some special functions for invoking Haskell functions from within Java. These functions are not publicly exported by the low level interface. They are however dynamically bound to native method stubs in the `HFunction` class, which is automatically loaded when the virtual machine is started up via `createVM`. The function `registerCallbacks` takes a reference to the `HFunction` class and registers the special functions with these native stubs.

registerCallbacks `Ptr JVM -> Ptr JClassRef -> IO Bool`

The exact mechanism for callbacks is discusses in section 4.4.5. The implementation of `registerCallbacks` is the following:

```
jboolean
registerCallbacks(vm_t* vmi, jclass hFunction) {
    JNIEnv* env = vmi->env;

    JNINativeMethod method;
    jint result;

    /* register native call function */
    DEBUG1("Registering call(...): %p", _callCallback)

    method.name = "call";
    method.signature = "(JLjava/lang/reflect/Method;[Ljava/lang/Object;)Ljava/lang/Object;";
    method.fnPtr = _callCallback;

    result = (*env)->RegisterNatives(env, hFunction, &method, 1);
    if (result < 0) {
        (*env)->ExceptionDescribe(env);
        return JNI_FALSE;
    }

    /* register native release function */
    DEBUG1("Registering release(...): %p", _releaseCallback)

    method.name = "release";
    method.signature = "(J)V";
    method.fnPtr = _releaseCallback;

    result = (*env)->RegisterNatives(env, hFunction, &method, 1);
    if (result < 0) {
        (*env)->ExceptionDescribe(env);
        return JNI_FALSE;
    }

    return JNI_TRUE;
}
```

4.3.12 Workarounds

There is one special function which is needed on Mac OS X if the programmer wants to create an AWT or Swing application:

`runCocoaMain :: IO ()`

Mac OS X has some special requirements on the threading of graphical applications, namely the Cocoa Event Queue has to be running on the main thread. It is therefore necessary to fork a thread, continue the application on this other thread, and start the cocoa event queue manually on the main thread. That is what this function does. It is worth noting that the function never returns, since naturally the event queue is implemented as an inifinite loop (this is why the programmer has to resort to another thread). The function is used on `runJavaGui` in the medium level interface and will not do anything on all other platforms. A further discussion of this topic is done in section 4.4.8.

The implementation of this function is:

```
void
runCocoaMain()
{
    #if defined(FFIJNI_MACOSX) && defined(FFIJNI_OSX_GUI)
    DEBUG("Retrieving NSApp...")
    void* clazz = objc_getClass("NSApplication");
    void* app = objc_msgSend(clazz, sel_registerName("sharedApplication"));

    DEBUG1("-> %p", app)

    DEBUG("Starting cocoa main runloop")
    objc_msgSend(app, sel_registerName("run"));
    #endif
}
```

4.4 Medium Level Interface

The medium level interface is located in a package `Foreign.Java`. It exports functions for looking up method IDs, field IDs, constructor IDs, without exposing the Java Native Interface directly, thus the programmer will not have to deal with pointers directly. Programming with the medium level interface is safe in the sense that no low level methods which could provoke a segmentation fault or the like are exposed.

4.4.1 Haskell extensions being used

While the low level interface is portable to all Haskell compilers that support the Foreign Function Interface, the medium level interface makes use of some extensions of the Haskell language. It is not necessary to do so, but greatly enhances the usability of the resulting API.

The extensions being used are:

MultiParamTypeClasses
Type classes with multiple parameters effectively describe relations between types. They are used in the medium level interface to ensure that a given Haskell function corresponds to the description of a given Java method. Examples of the use of multi parameter type classes can be seen in the contexts of `callMethod`, `bindMethod`, etc[24].

FunctionalDependencies
Multi parameter type classes are amlost unusable without some kind of assistance for the type inference engine. Functional depencies provide that assistance. Most prominently, functional dependencies can fix the type of one parameter of a type class by making it depend on another. This is used in the definition of the function `getMethod`, `callMethod`, `callMethodE`, etc[25].

FlexibleInstances
Many instance declarations can simply not be given without flexible instances, especially in the context of multi parameter type classes. Specifically flexible instances are needed to provide instances for method descriptors, see section 4.4.3[26].

FlexibleContexts
If you want to mention a flexible instance in a context of a function, instance, or type class declaration you need also flexible contexts[27].

UndecidableInstances
While the name is rather scary, this extension just lifts a constraint from the definition of type classes and class instances, which may lead to undecidable instances – i.e. it becomes possible to write instances of which it can not be decided whether there is one for a certain type or not. The bad thing that could

[24] GHC User's Guide, Version 7.6.2, § 7.6.1.1 Multi-parameter type classes
[25] GHC User's Guide, Version 7.6.2, § 7.6.2 Functional dependencies
[26] GHC User's Guide, Version 7.6.2, § 7.6.3.1 Relaxed rules for the instance head
[27] GHC User's Guide, Version 7.6.2, § 7.6.3.1 Relaxed rules instance contexts

happen is that the type checker is sent to an infinite loop. Since the java-bridge library does compile, this is obviously not the case[28].

These extensions are known to be supported not only by GHC, but also by Hugs[29] (in fact some of them originated there).

4.4.2 The Java Monad

As described in section 3.1.2 every action that is executed in the JVM should be encapsulated in a custom monad, the Java monad. The medium level interface defines that monad and offers the function `runJava`.

runJava `Java a -> IO a`

runJava' `[String] -> Java a -> IO a`

`runJava` will internally call `createVM` and `destroyVM` from the low level interface. `runJava'` uses `createVM'` instead and allows for some additional arguments to the JVM, for example:

```
runjava' ["-Djava.class.path=/java"] javaAction
```

This is the implementation of `runJava'`:

```
runJava' opts f = do

    str <- mapM newCString (augmentOpts opts)
    ptr <- newArray str
    vm  <- JNI.createVM' (fromIntegral $ length str) ptr

    mapM_ free str >> free ptr

    if vm == nullPtr then do
                            libjvmPath <- JNI.getLibjvmPath >>= peekCString
                            throw $ JvmException libjvmPath opts
                     else return ()

    (result, _) <- finally (runStateT (_runJava f) (newJVMState vm))
                           (JNI.destroyVM vm)

    return result
```

It is also possible to initialize the virtual machine once in a Haskell process. The JVM will then live as long as the Java process. When initialized using `initJava`, `runJava` will not automatically tear down the virtual machine after the last thread executed actions it:

initJava `[String] -> IO ()`

As mentioned in section 4.4.8, Mac OS X has some special requirements on GUI applications. It is therefore necessary to run the Java monad using `runJavaGui`. On all other platforms `runJavaGui` does exactly the same as `runJava`, it is therefore safe and still portable to always call `runJavaGui`.

runJavaGui `Java a -> IO ()`

runJavaGui' `[String] -> Java a -> IO ()`

[28] GHC User's Guide, Version 7.6.2, § 7.6.3.3 Undecidable instances
[29] The Hugs 98 User Manuel § 7 An overview of Hugs extensions

```haskell
import Foreign.Java

main = runJava $ do
    (Just threadClass) <- getClass "java.lang.Thread"
    currentThread <- threadClass `bindStaticMethod`
                       "currentThread" ::= object "java.lang.Thread"
    getName <- threadClass `bindMethod` "getName" ::= string

    (Just thread) <- currentThread
    (Just name) <- getName thread

    io$ putStrLn name
```

Figure 43: *Exemplary usage of* `runJava`

Implementation The Java monad is a State monad transformer that wraps the IO monad. The state of that monad is a value of type `JVMState`.

```haskell
newtype Java a = Java { _runJava :: StateT JVMState IO a }
  deriving (Monad, MonadState JVMState, Functor, MonadIO)

data JVMState = JVMState {
    -- | The actual pointer to the virtual machine.
    jvmPtr :: Ptr Core.JVM,

    -- some more fields. They actually only serve for the
    -- caching of frequently used method IDs and are
    -- omitted here for brevity.
}
```

Note that this definition requires the `GeneralizedNewtypeDeriving` extension. One can omit the usage of this extension and implement the `return` and `bind (>>=)` functions by hand (after all, it is a simple state monad).

4.4.3 Method and Field Descriptors

The low level interface requires the programmer to explicitly request method or field IDs using method and field descriptors that look like the internal format of method and field descriptors in class files, i.e. strings like
`(JJ[IZLjava/lang/Object;[Ljava/util/reflect/Type;)J` .

These method descriptors are highly inconvenient to use. Furthermore they do not feature any type safety, since in principle any string can be used to request an ID from the JVM. The medium level interface remedies this by introducing a dedicated datatype `MethodDescriptor`[30]. A `MethodDescriptor` is a value of the form `String ::= p` where `p` is the signature of a Java method. Signatures are represented as a value of the types `Z`, `C`, `B`, `S`, `I`, `J`, `F`, `F`, `L`, `V`, `A`, `X`, or `P`. These are translations from the low level method descriptors into the Haskell type system. Since these capital letters are a bit unintuitive aliases are introduced: `boolean`, `char`, #byte#, etc. It is worth noting that `A` and `X` actually do not exist in the JNI. `A` is a translation of `[` and denotes arrays, `X` is an alias for `L "java.lang.String"` and is needed for the special support for strings that is offered by the low level interface and of course by the medium level interface too.

The list of parameters to a function is church encoded, using `P` as `Cons` operator (*P* is for *pair*). `-->` is a more readable alias for `P`. It is now possible to give type safe method descriptors:

[30] Defined in Foreign.Java.Types and Foreign.Java.JNI.Types

```haskell
data MethodDescriptor p = String ::= p

(-->) :: a -> x -> P a x
a --> x = P a x

infixr 9 -->
infixl 8 ::=

data Z = Z          ; boolean = Z
data C = C          ; char    = C
data B = B          ; byte    = B
data S = S          ; short   = S
data I = I          ; int     = I
data J = J          ; long    = J
data F = F          ; float   = F
data D = D          ; double  = D
data L = L String   ; object  = L
data V = V          ; void    = V
data A x = A x      ; array   = A
data X = X          ; string  = X

data P a x = P a x
```

Figure 44: *The MethodDescriptor datatype.*

Name	**Low and Medium Level Method Descriptor**

main	`([Ljava/lang/String;)V`
	`"main" ::= array string --> void`

indexOf	`([Ljava/lang/Object;)I`
	`"indexOf" ::= array (object "java.lang.Object") --> int`

plus	`(LL)L`
	`"plus" ::= long --> long --> long`

Method descriptors are being used by the functions `getMethod`, `getStaticMethod`, `bindMethod`, and `bindStaticMethod`. The `getMethod`/`getStaticMethod` functions return a `JMethod p` or a `JStaticMethod p` respectively where `p` is the type of the method.

The functions `bindStaticMethod`, `bindMethod`, `callStaticMethod`, and `callMethod` all have a type like the following:

```haskell
MethodCall p b => p -> b
```

`p` is the type of the function, i.e the `p` in `String ::= p` or the `p` in `JMethod p`. `MethodCall` is a type class that takes two parameters with a functional dependency that deduces the type of `b` from `p`. This way the above functions will themselves generate functions that correspond to the given type `p`:

Method Descriptor (without Name)	**Type**	**Resulting Function**
object ”java.lang.String” $\longrightarrow$ int	P (L I)	Maybe JObject $\rightarrow$ Java Int32
char $\longrightarrow$ int $\longrightarrow$ string	P C (P I X)	Word16 $\rightarrow$ Int32 $\rightarrow$ Java String
void	V	Java ()
string $\longrightarrow$ object ”java.util.TreeMap”	P (X L)	String $\rightarrow$ Java (Maybe JObject)

It is worth noting that this mechanism discards the actual type of the object (i.e. `java.util.TreeMap` in the last example in the table above). The problem is that we can not deduce a type from the string that specifies the name of the class other then `[Char]`. We could of course create type level names like

```haskell
J (A (V (A (Dot (U (T (I (L (Dot (T_ (R (E (E (M_ (A (P Stop)))))))))))))))))
```

But that would hardly be useful at all[31].

[31] `http://hackage.haskell.org/package/names` – Albeit I have created a package which offers

4.4.4 References

The low level interface requires the programmer to deal with pointers and global references which have to be freed manual. Pitfalls like null pointers or unsafe array access lurk everywhere in the low level interface, just as they do in C. To prevent this from happening the medium level interface encapsulates pointers and references via `newtype` declarations so that the complexity of the underlying low level interface is hidden. Methods dealing with these new datatypes will only perform safe operations or provide methods for handling errors (other than segfaulting).

Specifically these new datatypes are[32]:

```haskell
-- | A reference to an arbitrary Object.
newtype JObject = JObject { jobjectPtr :: ForeignPtr JObjectRef }

-- | A reference to a Class object.
newtype JClass = JClass { jclassPtr :: ForeignPtr JClassRef }

-- | A reference to an Exception.
newtype JThrowable = JThrowable { jthrowablePtr :: ForeignPtr JThrowableRef }

-- | A reference to an Array in the JVM.
data JArray e = JArray {
    jarrayLength :: Int32,
    jarrayPtr :: ForeignPtr JObjectRef
  }
```

The functions in the medium level interface all guarantee that values of these types never represent a null reference. Furthermore all of the internal references are `ForeignPtr` instead of `Ptr`. `ForeignPtr` have an associated finalizer which will be triggered when the Haskell runtime system garbage collects the `ForeignPtr`.

4.4.5 Callbacks

For properly using a Java library from within Haskell (such as Swing) it is crucial to be able to pass callbacks from Haskell to Java. For example we would like to pass an `ActionListener` to the `addActionListener(ActionListener l)` method of a `javax.swing.JButton`. In Java this is usually done by creating an anonymous class:

```java
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // do Something
    }
});
```

We would like to do the same. In order to so so we will have to create an ActionListener. Fortunately Java does have a tool for this: *dynamic proxy classes.*

From the documentation on `java.lang.reflect.Proxy`:
A DYNAMIC PROXY CLASS *(simply referred to as a proxy class below) is a class that implements a list of interfaces specified at runtime when the class is created, with behavior as described below. [...] A* PROXY INSTANCE *is an instance of a proxy class.*[33]

We would like to pass a *proxy instance* of `ActionListener` to the `addActionListener` method. For doing so we need a `java.lang.reflect.InvocationHandler` that delegates the invocations to a Haskell function. Furthermore we need to export a Haskell function from the Haskell runtime so that it can be called from the outside.

The `InvocationHandler` interface specifies a single method:

type level names encoded like here which achieves usability by using the TemplateHaskell extension to create type level names from strings.

[32] Defined in Foreign.Java.Types

[33] Java Platform Standard Edition 6, java.lang.reflect.Proxy

`Object invoke(Object proxy, Method method, Object[] args)\.`

We can define an equivalent function in Haskell:

`invoke :: JObject -> JObject -> JObject -> Java (Maybe JObject)`

Unfortunately this function can not be exported by the FFI, since every type that is mentioned in a signature needs to be a primitive type (`Int32`, #Char#, . . .), a pointer (`Ptr`), or it must be isomorphic to one of these types (`newtype X = X Int32`).

If we use the low level interface instead we can create a function that can be exported by the FFI:

```
invoke :: Ptr JObjectRef -> Ptr JObjectRef -> Ptr JObjectRef -> IO (Ptr JObjectRef)
```

This function is also returning a result in the IO monad instead of the Java monad. This is necessary since the allowable return types for functions that are exported by the FFI are only the types which can be used as arguments or any of these types encapsulated in the IO monad.

We will refer to the type of this function as `WrappedFunc` as we are going to pass it through a *wrapper* function:

```
type WrappedFunc = Ptr JObjectRef -> Ptr JObjectRef -> Ptr JObjectRef -> IO (Ptr JObjectRef)
```

This function can be exported by a *dynamic wrapper*[34]:

```
foreign import ccall safe "wrapper"
    wrap :: WrappedFunc -> IO (FunPtr WrappedFunc)
```

In order to dynamically implement a Haskell interface using a proxy class we need to pass a wrapped function to a special Java class that creates a *proxy instance* from this function. We are going to call this class `HFunction`. This class will also implement the `InvocationHandler` and doing the actual calling back.

The `HFunction` class looks like this:

```
import java.lang.reflect.*;

/**
 * An HFunction can be used as a invocation handler which actually
 * handles the invocation by calling a Haskell function.
 *
 * @author Julian Fleischer
 */
public class HFunction implements InvocationHandler {

    private final long hFunction;

    static native void release(long func);
    static native Object call(long func, Method self, Object[] args);

    /**
     * Creates an HFunction from a function pointer.
     *
     * @param func
     */
    public HFunction(long func) {
        hFunction = func;
    }

    public Object invoke(Object proxy, Method method, Object[] args) {
        return call(hFunction, method, args);
    }

    protected void finalize() {
        release(hFunction);
```

[34] Haskell 2010 Language Report § 8.5.1 Standard C Calls – dynamic wrapper

```
    }

    /**
     * Make an HFunction for a given iface from a function pointer.
     *
     * @param iface
     * @param func
     */
    @SuppressWarnings("unchecked")
    public static <T> T makeFunction(Class<T> iface, long func) {
        InvocationHandler handler = new HFunction(func);
        return (T) Proxy.newProxyInstance(
                            iface.getClassLoader(),
                            new Class<?>[] { iface },
                            handler);
    }
}
```

A shortcoming of Java and the JNI is that there is no way of representing a function pointer (whereas both C and Haskell have such a thing – `FunPtr` in Haskell). It is therefor necessary to cast the function pointer obtained by `wrap` into an integer and pass that integer to the virtual machine. This is considered bad practice in the C community and it is not covered by the C standard (functions and data could be separated and have completely different implementations, for example in a machine with *Harvard architecture*).

Being able to cast integers to function pointers is however covered by the POSIX standard, and it is also guaranteed to work in Windows. It is in fact part of the Win 32 API, for example for loading dynamic libraries. Here is the part of the java-bridge which dynamically loads the `libjvm` library (in Windows):

```
HINSTANCE hVM = LoadLibrary(_libjvm_path);
if (!hVM) {
    return -41;
}
_GetCreatedJavaVMs =
    (jint (*)(JavaVM**,jsize,jsize*))
    GetProcAddress(hVM, "JNI_GetCreatedJavaVMs");
_CreateJavaVM =
    (jint (*)(JavaVM**,void**,void*))
    GetProcAddress(hVM, "JNI_CreateJavaVM");
```

This snippet requires casting integers to a function pointers (the very same example could be given in POSIX, looking only slightly different).

Now that we have put an end to all uncertainty at this point, we can move on and create a function `implementInterfaceBy`[35] which uses a function like the first proposal for an invoke function, creates a `WrappedFunc` from it, wraps it using `wrap_` and passes it to `HFunction.makeFunction`, which returns a `proxy instance` for the class specified by its name:

```
type InterfaceFunc = JObject -> JObject -> JObject -> Java (Maybe JObject)

foreign import ccall safe "wrapper"
    wrap_ :: WrappedFunc -> IO (FunPtr WrappedFunc)

implementInterfaceBy :: String        -- ^ name of the interface to be implemented
                     -> InterfaceFunc -- ^ implementation for @invoke@
                     -> Java JObject  -- ^ A proxy instance
implementInterfaceBy ifaceName func = do
    iface <- getClass ifaceName >>= asObject . fromJust
    (Just clazz) <- getClass "HFunction"
    success <- registerCallbacks clazz
    if success then return () else fail "JNI␣native␣methods␣could␣not␣be␣registered"
```

[35] Defined in and exported by Foreign.Java.Bindings

```
    makeFunction <- clazz `bindStaticMethod` "makeFunction"
        ::= object "java.lang.Class" --> long --> object "java.lang.Object"
    (Just impl) <- io (intify func) >>= makeFunction (Just iface)
    return impl
  where
    wrap :: InterfaceFunc -> IO (FunPtr WrappedFunc)
    wrap f = do

        let proxyFunc vm self method args = do
                self'   <- Core.JObject <$> newForeignPtr JNI.release self
                method' <- Core.JObject <$> newForeignPtr JNI.release method
                args'   <- Core.JObject <$> newForeignPtr JNI.release args

                jobj <- runJava vm (f self' method' args')

                case jobj of
                    Nothing -> return nullPtr
                    Just (Core.JObject ptr) -> withForeignPtr ptr return

        wrappedFunc <- wrap_ proxyFunc

        return wrappedFunc

    -- Wrap a frunction and cast the resulting FunPtr to an Int64 (jlong)
    intify :: InterfaceFunc -> IO Int64
    intify = fmap (fromIntegral . ptrToIntPtr . castFunPtrToPtr) . wrap

    runJava :: Ptr Core.JVM -> Java a -> IO a
    runJava vm f = runStateT (_runJava f) (newJVMState vm) >>= return . fst
```

Note that this implementation already uses `getClass` and `bindStaticMethod` from the medium level interface and turns `InterfaceFunc` functions (which are in the style of the medium level interface) into proxy instances, not a `WrappedFunc` – `WrappedFunc` is only used as an intermediate.

A remark on invoke: The alert reader may wonder why a `WrappedFunc` has three arguments while the `call` method in `HFunction` has only two arguments. The third argument (actually the first, but nevertheless an additional argument) is given by the JNI and references the calling class (if a native static function was called) or the calling object (if a native non-static function was called). The JNI also gives a pointer to the instance of the invoking JVM, which is why there is a new `runJava` function defined within `implementInterfaceBy`, which executes the action in the Java monad with that pointer.

`implementInterfaceBy` will also be used to allow Java classes to call Haskell code. This is discussed in section **??**.

Using our new asset we can finally pass an `ActionListener` to `addActionListener`:

```
runJava $ do
    (Just jbuttonClass) <- getClass "javax.swing.JButton"
    (Just jbutton) <- newObject jbuttonClass

    addActionListener <- jbuttonClass `bindMethod`
        "addActionListener" ::= object "java.awt.event.ActionListener" --> void

    let handler :: InterfaceFunc
        handler _hfunc _method args = do
            -- handle the event, args contains an array
            -- with arguments, of which the first is an
            -- @ActionEvent@
            return ()

    implementInterfaceBy "java.awt.event.ActionListener" handler
        >>= addActionListener jbutton
```

4.4.6 Garbage Collection

The java-bridge library makes use of the Haskell 2010 type `ForeignPtr`. These pointers are pointers with an associated finalizer. We have already seen the finalizer in the description of the low level interface (`release`).

Whenever a function in the medium level interface retrieves a pointer from the virtual machine, it creates a `ForeignPtr` from it and returns that, encapsulated in a new type which hides the actual implementation. This way the programmer is freed from the burden of taking care of releasing references, the Haskell runtime system will do it.

Releasing function pointers is also necessary. In section 4.4.5 the `HFunction` class was presented which is a Java equivalent for Haskell's `FunPtr` (which the Java standard library unfortunately lacks). Java classes can have a `finalize` method which is called when the garbage collector is freeing a class. The `finalize` function in `HFunction` will call the native static method `release` in the HFunction class which takes as argument a function pointer (cast to a long). `release` will have been registered by `implementInterfaceBy` (page 47) via `registerCallbacks` (page 40). `release` is registered with a function that is exported by the Haskell part of the java-bridge library, `freeFunPtr`[36]:

```
foreign export ccall freeFunPtr :: FunPtr WrappedFunc -> IO ()

freeFunPtr :: FunPtr WrappedFunc -> IO ()
freeFunPtr ptr = freeHaskellFunPtr ptr
```

`freeHaskellFunPtr` is provided by the Haskell FFI[37].

4.4.7 Concurrency

It is worth noting that programs that use the java-bridge library need to be compiled using a threaded Haskell runtime (In GHC this is induced by the `-threaded` switch).

Threads can be started within the Java monad using the `forkJava` method. While in principle you could also use the functions provided by `Control.Concurrent`, lifted into the Java monad using `liftIO`, I recommend using the `forkJava` function. It provides a clean and fail safe implementation for starting a thread in Haskell in the presence of a virtual machine. The important thing is, that `forkJava` uses `forkOS`, which spawns a *bound* thread. Threads in concurrent Haskell are lightweight threads which can be moved from one OS thread to another. Using lightweight threads may lead to conflicts between the Haskell runtime and the Java virtual machine.

A Java method might be executed in one thread that uses for example Javas `ThreadLocal` storage. If another method is invoked which tries to access that same storage, but from a different thread (the Haskell runtime might have moved the invoking lightweight thread to another OS thread) the thread local state will be lost (at least it is momentarily not accessible by this method).

An implementation for `forkJava` and `waitJava` is:

```
newtype JavaThreadId a = JavaThreadId (MVar (Either SomeException a))

forkJava :: Java a -> Java (JavaThreadId a)
-- ^ A utility function for forking an OS thread which runs in the
-- Java Monad. It will return a 'JavaThreadId' which you can wait on
-- using 'waitJava'.
forkJava t = io $ do
    lock <- newEmptyMVar
    _ <- forkOS $ do
        result <- try $ runJava t
        putMVar lock result
```

[36] Defined in and exported by Foreign.Java.Bindings
[37] Defined in Foreign.Ptr

```haskell
    return $ JavaThreadId lock

waitJava :: JavaThreadId a -> Java (Either SomeException a)
-- ^ Wait for a Java Thread to exit. If the thread exits abnormally
-- (that is, if an exception occurred), this function will return
-- @Left SomeException@. Otherwise it will return the result of the
-- computation as @Right a@.
waitJava (JavaThreadId mvar) = io $ takeMVar mvar
```

Thread safety of function calls between runtimes is already provided by the Haskell FFI. The foreign imports that access the C functions of the java-bridge library are marked as `safe`, which tells the Haskell compiler to ensure that concurrent access is correctly synchronized with garbage collection etc. It is however only activated when compiled using a threaded runtime. If not, the virtual machine might invoke a Haskell function concurrently and thereby crash the runtime system of Haskell.

Concurrent calls from a Haskell process into the virtual machine are safe in the sense that they do not crash the virtual machine. In all other aspect they work like an ordinary concurrent invocation from an arbitrary Java thread, hence the same problems as in ordinary Java programming may arise.

4.4.8 OS X Issues

Graphical User Interfaces are implemented via Cocoa on Mac OS X. The port of AWT to Mac OS X is known as *CocoaAWT*. Both Cocoa and AWT do have distinct event queues. Cocoa has a requirement on the event queue which CocoaAWT inherits: The Cocoa event queue has to be running on the first thread[38]. If a Haskell process starts up, it will be the process running on the main thread. Initially it will be the only thread running.

When starting up a virtual machine, the start up sequence is initiated on the main thread, but new threads will be created which are not the main thread. For example the JVM starts a thread for catching uncaught exceptions, another thread for garbage collection, etc.

Both the AWT event queue as well as the Cocoa event queue are initially not running. When the first AWT action is executed in the JVM, the AWT event thread (the thread which handles the event queue) will be started up. If that first AWT action was issued by the main thread, CocoaAWT will detect that it has been invoked from the main thread where the Cocoa event queue should be running. CocoaAWT will therefore do the right thing and fail with an error:

```
wing[1883:707] Cocoa AWT: Apple AWT Java VM was loaded on first thread -- can't start AWT.
    0   liblwawt.dylib       0x0000000117e87ad0 JNI_OnLoad + 468
    1   libjava.dylib        0x00000001026076f1      Java_java_lang_ClassLoader_00024[...]
    2   ???                  0x000000010265af90 0x0 + 4335185808
)
```

The solution is to manually initialize the AWT system on a thread other than the main thread. This will at least render a window – and freeze. The reason for the freeze is that CocoaAWT assumes that a Cocoa event queue is already running on the main thread (Starting the Cocoa event queue normally is the very first thing a graphical application does). It therefore waits for events from the Cocoa event queue (events are handled by Cocoa and than delegated to CocoaAWT which handles them on the AWT event queue) which is not running at all! Thanks to the extremely dynamic nature of the Objective-C runtime (Cocoa is an Objective-C framework) we do not get an error or anything, the application just freezes as its request for gets lost and the AWT event queue will be blocked indefinitely.

[38]`http://developer.apple.com --` – Cocoa Threading Programming Guide, Version 2010-04-28

To ultimately solve the problem we have to do initialize the Cocoa event queue ourselves. The initialization of a the event queue is normally done when a Cocoa application starts up. The ”Apple AWT Java VM” obviously does not do this by itself but assumes that this has already been done (probably since it is being invoked via libjvm instead of in standalone process). Therefore we try to be a good host and create the AWT event queue:

```
void* clazz = objc_getClass("NSApplication");
void* app = objc_msgSend(clazz, sel_registerName("sharedApplication"));
objc_msgSend(app, sel_registerName("run"));
```

This piece of code is provided by the `runCocoaMain` function which was presented in the low level interface.

Since the Cocoa event queue will occupy the main thread once its started, we have to resort to another thread. Therefore a `runJavaGui'` function is defined in the medium level interface, which transparently does that for us:

```
runJavaGui' opts java = runJava' opts $ do
    _ <- forkJava java
    io JNI.runCocoaMain
```

4.4.9 Exception Handling

Each of the functions `callMethod`, `callStaticMethod`, `newObject`, and `newObjectFrom` come on three versions: as (using `callMethod` exemplary) `callMethod`, `callMethodE`, and `callMethodX` (i.e. plain, suffixed with `X`, and suffixed with `E`). The JNI requires that the programmer manually checks for exceptions, using `exceptionCheck` (see low level interface). Both the plain version as well as the functions suffixed with `E` will do this check. The plain version will call the Java monads `fail` method if an exception occurred (i.e. re-raise the exception). The `E` functions on the other hand always have the ultimate return type of `Java (Either JThrowable a)` where `a` is the type which would be returned by the plain version.

The `X` methods do not check for exceptions and should only be used if the programmer is sure that the invoked method will definitely not raise an exception.

```
-- Will @fail@ in case of an exception.
(Just thread) <- callStaticMethod currentThread

-- thread' :: Either JThrowable (Maybe JObject)
thread' <- callStaticMethodE currentThread
```

4.4.10 Retrieving Classes, Fields, and Methods

getClass `(Monad m) => String -> Java (m JClass)`
Find a Java class, return `Nothing` it the class does not exist (Actually this function can be used with any monad, it will call the `fail` function if the class does not exist, which in case of the `Maybe` monad would be `Nothing`).

getConstructor `(Monad m) => JClass -> a -> Java (m (JConstructor a))`
Retrieve a constructor ID (encapsulated in a #JConstructor#) or fail (e.g. Nothing) if there is no constructor with the given signature.

newObject `JClass -> Java (Maybe JObject)`
Create an object using the default constructor for a given class. If that constructor is not accessible, `fail` (e.g. Nothing).
Example: `newObject stringClass >>= maybe (...fail...) (\x -> ...success...)`.

newObjectE `JClass -> Java (Either JThrowable (Maybe JObject))`
`E` variant of `newObject`.

newObjectX `JClass -> Java (Maybe JObject)`
`X` variant of `newObject`.

newObjectFrom	`NewObject p b => JConstructor p -> b`	Creates an object from a previously retrieves constructor.
newObjectFromE	`NewObjectE p b => JConstructor p -> b`	`E` variant of `newObjectFrom`.
newObjectFromX	`NewObjectX p b => JConstructor p -> b`	`X` variant of `newObjectFrom`.
getMethod	`JClass -> MethodDescriptor p -> Java (Maybe (JMethod p))`	Retrieve a method id from the Java monad. `fail` if the method does not exist or can not be retrieved (e.g. `Nothing`).
getStaticMethod	`JClass -> MethodDescriptor p -> Java (Maybe (JStaticMethod p))`	Retrieve a static method id from the Java monad. `fail` if the method does not exist or can not be retrieved (e.g. `Nothing`).

`bindMethod` and `bindStaticMethod` are convenience functions that can be used to directly retrieve a Haskell function `b` from a `MethodDescriptor p`. The will `fail` in the Java monad (i.e. raise an ioError) if the method can not be retrieved, while `getMethod` and `getStaticMethod` will only fail in the monad `m` (which may be the Maybe monad, or the error monad).

bindMethod	`MethodCall p b => JClass -> MethodDescriptor p -> Java (Maybe (JObject -> b))`
bindStaticMethod	`MethodCall p b => JClass -> MethodDescriptor p -> Java (Maybe (JObject -> b))`

4.4.11 Calling Methods

Functions for calling methods.

callMethod	`MethodCall p b => JMethod p -> JObject -> b`	Call a method. `JMethod p` must be retrieved using `getMethod`. `p` determines the type of the function `b`.
callMethodE	`MethodCallE p b => JMethod p -> JObject -> b`	`E` variant of `newObjectFrom`.
callMethodX	`MethodCallX p b => JMethod p -> JObject -> b`	`X` variant of `newObjectFrom`.
callStaticMethod	`StaticCall p b => JStaticMethod p -> JObject -> b`	Call a static method. `JStaticMethod p` must be retrieved using `getStaticMethod`. `p` determines the type of the function `b`.
callStaticMethodE	`StaticCallE p b => JStaticMethod p -> JObject -> b`	`E` variant of `newObjectFrom`.
callStaticMethodX	`StaticCallX p b => JStaticMethod p -> JObject -> b`	`X` variant of `newObjectFrom`.

4.4.12 Reading and Writing Fields

Functions for reading and writing fields.

getField	`Param a => JClass -> String -> a -> Java (Maybe (JField a))`
getStaticField	`Param a => JClass -> String -> a -> Java (Maybe (JStaticField a))`
readField	`Field a b => JField a -> JObject -> Java b`
writeField	`Field a b => JField a -> JObject -> a -> Java ()`
readStaticField	`Field a b => JStaticField a -> Java b`
writeStaticField	`Field a b => JStaticField a -> a -> Java ()`

4.4.13 Utlity Functions

isInstanceOf	`JObject -> JClass -> Java Bool`	Checks whether the `JObject` is an instance of the given `JClass`.

4.4.14 Interaction with the IO Monad

io `IO a -> Java a`
Execute an IO action within the Java monad.

```
runJava $ do
    ... java ...
    io $ putStrLn "Haskell␣IO"
    ... java ...
```

forkJava `Java a -> Java (JavaThreadId a)`
This function forks a Java computation in a new Haskell thread and returns a `JavaThreadId`. This is not a thread id by which Java knows a thread, but an object which can be used inside the Java monad to wait on a thread using `waitJava`.

waitJava `JavaThreadId a -> Java (Either SomeException a)`
Waits for a thread forked using `forkJava`. If an exception occurred in that thread, it is returned as `Left SomeException`. If not, the result of type `a` is returned.

4.5 Summary

In this section the design and implementation of the `java-bridge` library was presented. The library comprises two interfaces, a low level interface and a high level interface.

The low level interface is a straight-forward binding to the C API that is offered by the Java Native Interface. It also introduces some convenience functions for dealing with strings and arrays and. It also offers some functions which per se have no value to the programmer, but which are being used by the medium level interface to define some more complex constructs.

The programmer is however responsible for checking for exceptions and managing references. The API is also rather clumsy, since it requires the programmer to create method descriptors like `(JJ)Ljava/lang/String;` and deal with #nullPtr# values.

The medium level interface is a more convenient alternative to the low level interface. It automatically manages references by using Haskell's `ForeignPtr`s. It will also hide every sign of C, i.e. the programmer will only have to deal with well known datatypes and `JObject`s, `JClass`s, etc., but not with `Ptr`, `nullPtr` values, or the like. By the means of several Haskell extensions a greater type safety is achieved, you can for example not call a function retrieved by `"meth" ::= int --> long --> void` with 3 arguments (which is totally possible in the low level interface by passing a malformed `JValues` pointer). Also the medium level interface offers a tiny DSL for discovering methods, constructors, and fields.

It is however by no means an adequate translation of every possible Java interface. For example it is still possible to call a method on an incompatible object or to pass an incompatible object as an argument which expects a specific object type. Also it requires the programmer to lookup method, constructor, and field IDs prior to using them, which results in huge sections of boiler plate code.

A high level interface was presented in section 3, i.e. a translation of Java classes into Haskell functions with adequate types that preserve type safety and subtyping. Such an interface demands that some extra glue code is generated which comprises the translated version of a Java class. For the purpose of automatically creating such high level bindings to Java a tool, `j2hs` has been built, which is presented in the next section.

A haskell-bridge library? The reader has surely noticed that no effort so far has been undertaken to call Haskell from within Java (except for to call *back* from Java). This is due to the fact that Haskell functions can not be easily accessed by the outside world. Through the JNI *every* public Java function can be accessed, whereas Haskell functions have to be *explicitly* exported using an `export` statement. While the Foreign Function Interface allows for exporting Functions as `FunPtr`s (and even `StablePtr`s to data that is allocated on the Haskell heap), it is not possible to access arbitrary functions.

Accessing Haskell from another language will therefore always require some glue code to be generated which exports the functions that should be available in that other language.

5 The j2hs Bindings Generator

The `j2hs` bindings generator is a tool that is capable of converting the public API of Java classes to a Haskell representation which can be used to work with Java libraries from within Haskell.

5.1 Prerequisites

First and foremost the bindings generator will have to be able to read the interface of the given Java classes. Fortunately the Java Standard Platform offers the *Java Reflection API* which can be used for doing exactly that. The `java-bridge` which was presented in the last section can be used to access the *Java Reflection API*.

The Java Reflection API is a very clean API for gathering information about Java classes and we do not intend to discuss it any further, as this is beyond the scope of this thesis. We assume a certain familiarity of the reader with the Java Reflection API, if not, Suns website offers fairly good introduction to the topic (after all it is just another API).

5.2 Design

The bindings tool should be able to completely translate a given set of Java classes or Java packages using the translation scheme that was proposed in section 3. For doing a complete translation it will have to analyze and translate not only the given classes or packages but also any packages that are referenced by them.

The bindings generator should thus be able to not only reflect a Java class, but also to find the dependencies of such a class, and apply that procedure exhaustively.

The target of the translation should be a set of Haskell modules that represent the given Java classes and packages and their dependencies.

5.2.1 Circular References

A straight-forwards translation of a Java API would be to create exactly one module for each package. Such a module would however tend to get rather big and there would be many overlapping identifiers, since many objects may define methods with the same name. A better approach is therefore to create a Haskell module for each Java class, which contains the datatype definitions as well as the function declarations for this Java class.

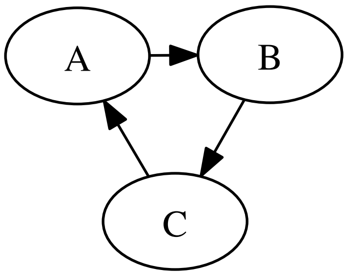
Unfortunately this approach still comes with a problem: Classes in Java may reference each other mutually. That is, a class `A` may import a class `B` and a class `B` may import a class `B`. While this should be the case in Haskell too (the langauge report states in Chapter 5 "Modules may be mutually recursive."[39]), it actually is not so. No Haskell compiler that the author knows of supports mutually recursive modules or whose imports form a cycle (though there are some work arounds, like `hs-boot` files, non of them cover the problem completely).

Fortunately the inheritance graph of Java classes is a Tree, i.e. does not contain any cycles. It is therefore at least possible to compile the typeclass and datatype declarations. It turns out that this solves our problem completely, all we have to do is to split up the modules which represent a Java class into to modules: One that describes the types and their inheritance relationships, and another one that actually implements their functions. Since function signatures only ever mention types and no other functions they will not require importing any other module that contains functions, only modules that contain types. Modules that contain types on the other hand can be compiled without any knowledge about their functions and they will never form a cycle.

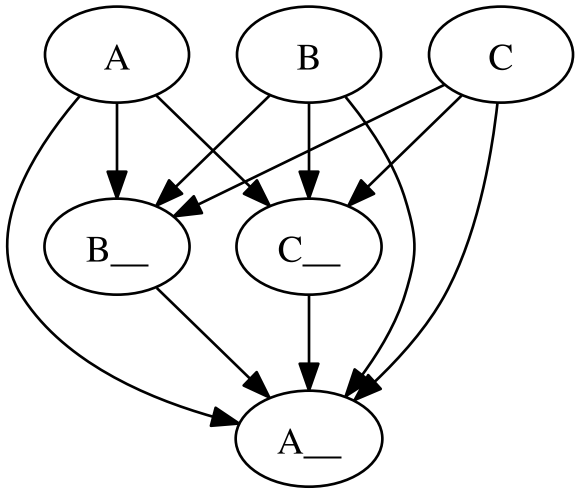
[39]Haskell 2010 Language Report § 5 Modules

Example: The classes A, B, and C have mutual dependencies. The translated Haskell modules could not be compiled, since the module imports would form a cycle.

```
public interface A {
    A a1();
    B b1();
    C c1();
}
public interface B extends A {
    A a2();
    B b2();
    C c2();
}
public interface C extends A {
    A a3();
    B a3();
    C a3();
}
```



If we split up the type class declarations into modules A, B, and C, and leave the functions in modules A, B, C, than we do not have a problem with circular dependencies:



It is now also possible to create modules for Java packages that comprise the type declarations of the classes contained in a package, since every type name exists only once in a package. We can just not import the modules. This is even a good thing, as it will allow the programmer to write imports like this:

```
import Java.Lang
import qualified Java.Lang.Object as Object
import qualified Java.Lang.String as String
```

The first import imports types and typeclasses `Object`, `Object'`, `String`, `String'`, etc. while the second two imports import the functions of these methods. Since both `Object` and `String` have a `toString` method we are able to differentiate the two by using qualified imports:

```
main = runJava $ do
    (Just obj) <- new'Object
    (Just str) <- new'String

    (Just str1) <- Object.toString obj
    (Just str2) <- String.toString obj
```

Though we have to admit that this is a rather artifical example, since `toString` is actually defined in the class `Object` and since the JVM performs late binding (toString is a virtual method and therefore `Object.toString` will invoke `String`'s `toString` method).

5.2.2 Dependency Hell

The `j2hs` library should be able to translate the Java standard library that accompanies Java SE 6 or Java SE 7. The Java standard library comprises about 3300 classes in more than 180 packages. Due to the problems presented in the previous section this will result in roughly 7000 modules to be created. Trying to compile this is practically impossible since the memory that GHC needs to compile such a lot of modules amount to several gigabytes (whether this can be considered a bug in GHC or not is debatable). However, even if GHC would not blow your memory, there is a fairly technical problem with compiling 7000 modules in one go: Neither GHC nor `cabal-install` (a Haskell build tool) support compiling such a huge amount of classes. Cabal for example silently exits with exitcode `127`.

Therefore we need to reduce the number of packages the number of packages that can be compiled in one go. Surely the Java standard library can be clustered into smaller components that are independent from each other? It would be rather nice to simply compile one package at a time. Figure 45 shows the dependencies between the packages of Java SE 6.

In the left graph the packages are arranged in a circle. The three dense corners – if it is sensible to speak about corners in a circle – are the packages `java.lang`, `java.util`, and `java.io`. The black arc on the left is Swing.

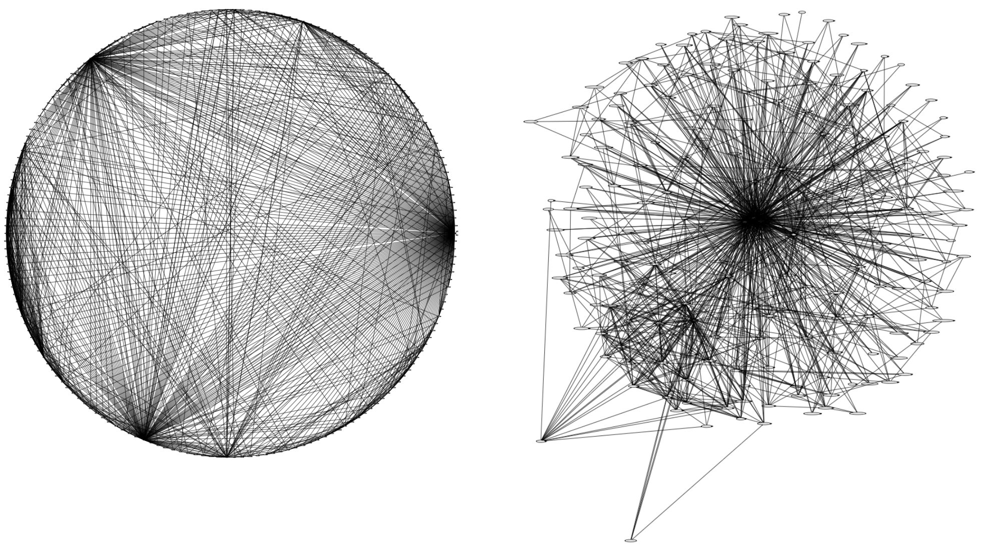


Figure 45: *Dependencies between Java packages.*

The right graph shows the same dependencies, laid out in a different way (not forced in a circle). The situation looks rather bad.

In fact if you take all packages that a single package depends on, directly or indirectly (i.e. all nodes that are reachable in the dependency graph), do this for all packages and create the intersection of the resulting sets, that leaves you with a core of 89 packages comprising about 1800 classes.

So, dividing among *packages* seems not to buy us much.

A solution is to identify the strongly connected components in the *class* dependency graph and sort them topologically. In the Java SE 7 for example the largest component in this list counts 63 nodes, with `java.lang.String` and `java.lang.Object` among them.

The algorithm implemented in the j2hs will do the following:

1. Exhaustively find all dependencies for the classes that are to be translated, using the Java Reflection API and the java-bridge to access the JVM;
2. build the dependency graph of all the classes that were found.
3. Identify the strongly connected components; (for example by applying Tarjan's strongly connected components algorithm);
4. Sort them topologically;
5. Group the resulting list into segments that are not mutually recursive (this is done as to create compilable packages of Haskell modules which are not too huge but also not to small, the latter would otherwise result in very many packages).

The segment size for the last step can actually be configured using j2hs. By applying this procedure the 3300 classes from Java SE 7 which are to be translated into nearly 7000 modules can be grouped into four packages of about 840 modules each, which can be compiled by cabal and GHC without any hassle.

Here is a sample run of the `j2hs` tool – without showing the generated sources of course (the `-X` flags advises `j2hs` to attempt to translate the whole standard library available on the host platform):

```
>>> ./dist/build/j2hs/j2hs -X -s840 -pcabal.tpl
Initializing JVM... Done.
Gathering reflection information... Done
    (found 3341 classes).
Attempting to find clusters... Done
    (found 2897 strongly connected components in 752 ranks).
Identified 4 segments of lengths (841, 841, 842, 817).
Generating modules for 841 classes in /j2hs/bindings/1... Done.
Generating modules for 841 classes in /j2hs/bindings/2... Done.
Generating modules for 842 classes in /j2hs/bindings/3... Done.
Generating modules for 817 classes in /j2hs/bindings/4... Done.
Generating modules for 184 packages in /j2hs/bindings... Done.
```

In this example run of `j2hs` 5 cabal packages were created. 4 packages comprising 841, 841, 842, and 817 classes and a meta package containing package modules. The seconds package dependes on the first, the third on the first and the second, etc., and the packages-package depends on all of the first four packages – but, no circular dependencies are created between packages (cabal packages may also not contain circular dependencies).

5.3 Implementation Details

Up to this point we have discusses (1) a translation scheme for expressing Java interfaces in Haskell, (2) the java-bridge library for accessing Java methods from within Haskell and calling back into Haskell, (3) the necessary prerequisites for building a high level bindings generator.

In section 3 we have left out questions concerning implementation details, while in section 4 we have focused on implementation details very much. The only question that is left is how the actual implementation of the proposed Haskell interface that results from the translation of a Java API looks like and how it is connected with the `java-bridge` library. This will be addressed in the following subsections.

The actual translation of a Java API will be done as proposed in section 3. That is, names are translated as discussed, and so functions and data types are created just like described:

- For each Java type, a type class is generated which acts as some kind of *marker interface* which can be used to express that a specific type actually represents a Java type that can be used with the interface (as in API) of the originating Java definition.

```
{- | @virtual equals(java.lang.Object) -> boolean@ -}
equals :: (JNIS.BooleanResult boolean,
           Java.Lang.Object__.Object this,
           Java.Lang.Object__.Object v1)
       => this -> v1 -> JNI.Java boolean
equals this a1 = do
    (P.Just clazz) <- JNI.getClass "java.lang.Object"
    (P.Just method) <- clazz `JNI.getMethod` "equals" JNI.::=
        JNI.object "java.lang.Object" --> JNI.boolean
    this' <- JNI.asObject this
    a1' <- P.Just <$> JNI.asObject a1
    result <- JNI.callMethodE method this' a1'
    JNIS.toBooleanResult result

{- | @virtual native final getClass() -> java.lang.Class@ -}
getClass :: (Java.Lang.Object__.Object this,
             JNIS.ObjectResult (object (Java.Lang.Class__.Class' v2)))
         => this -> JNI.Java (object (Java.Lang.Class__.Class' v2))
getClass this = do
    (P.Just clazz) <- JNI.getClass "java.lang.Object"
    (P.Just method) <- clazz `JNI.getMethod` "getClass" JNI.::=
        JNI.object "java.lang.Class"
    this' <- JNI.asObject this
    result <- JNI.callMethodE method this'
    JNIS.toObjectResult result

{- | @virtual native hashCode() -> int@ -}
hashCode :: (JNIS.IntResult int,
             Java.Lang.Object__.Object this)
         => this -> JNI.Java int
hashCode this = do
    (P.Just clazz) <- JNI.getClass "java.lang.Object"
    (P.Just method) <- clazz `JNI.getMethod` "hashCode" JNI.::= JNI.int
    this' <- JNI.asObject this
    result <- JNI.callMethodE method this'
    JNIS.toIntResult result
```

Figure 46: *A sample of the code generated by j2hs: equals(Object), getClass(), and toString().*

- For each Java type, a `newtype` declaration is generated as discussed in section 3.1.4. The `ObjectReference` in this data constructor is actually the type `JObject` which we have seen in the discussion of the `java-bridge` library.
- The implementation of a function is a do block which retrieves the arguments to the function via `toInt`, `toLong`, `toChar` etc. as discussed in section 3.1.3. These type classes and the instances for the primitive types are given in a supporting library that will be offered as part of the java-bridge library (in the module `Foreign.Java.Bindings`). The documentation to this Haskell module can be found in the appendix.

- A functions implementation will have to lookup its corresponding Java method id using the functions from the java-bridge library and invoke it accordingly. The result can than be converted to the requested return type using a polymorphic return type, as discussed in 3.1.9. The supporting type classes `ObjectResult`, `IntResult`, etc. can be found in the support module `Foreign.Java.Bindings`.

5.3.1 The Support Module Foreign.Java.Bindings

In order to support the high level bindings, the java-bridge library is extended by the module `Foreign.Java.Bindings`. This module comprises type classes for the conversion between Haskell types and different Java types:

```
class JBoolean a  where toBoolean :: a -> Java Bool
class JChar a     where toChar    :: a -> Java Word16
class JByte a     where toByte    :: a -> Java Int8
class JShort a    where toShort   :: a -> Java Int16
class JInt a      where toInt     :: a -> Java Int32
class JLong a     where toLong    :: a -> Java Int64
class JFloat a    where toFloat   :: a -> Java Float
class JDouble a   where toDouble  :: a -> Java Double
```

These type classes are actually used in the translation, as proposed in section 3.

Instances for these type classes are declared for the primitive types `Char`, `Int8`, `Word16`, `Int16`, `Int32`, `Int64`, `Double`, and `Float` where appropriate.

These type classes can be extended (type classes in Haskell are open) for example by providing an instance of `JInt` for the type `Java.Lang.Number'` or by providing an instance of the type class `Java.Lang.String` for the Haskell type `[Char]`.

Here is an example how both `Char`, `Int8`, and `Word16` can be treated as a Java `char`:

```
instance JChar Char      where toChar = return . fromIntegral . fromEnum
instance JChar Int8      where toChar = return . fromIntegral
instance JChar Word16    where toChar = return
```

This way it is possible to conventiently extend the automatic marshalling and unmarshalling mechanism that is provided by these type classes, even without recompiling of existing bindings.

These type classes are only used for the argument types. The result types are a bit more complex (we do not give all class declarations for brevity, in the last declaration just replace `<Type>` by `Boolean`, `Char`, `Byte`, #Short#, etc. and `<Result>` by `Bool`, `Word16`, `Int8`, `Int16`, etc.):

```
class ObjectResult m where
    toObjectResult :: Either JThrowable (Maybe JObject) -> Java m
class VoidResult m where
    toVoidResult :: Either JThrowable () -> Java m
class <Type>Result m where
    to<Type>Result :: Either JThrowable <Result> -> Java m
```

It is possible to extend the results types in much the same way as the argument types. For example it is possible to declare the an instance of the `ObjectResult` type class for a type `AsString` which can be used to marshal any object into a String (for example using its `toString` method):

```
newtype AsString o = AsString [Char]
    deriving (Eq, Show, Ord)
instance ObjectResult (AsString a) where
    toObjectResult = fmap AsString . either (\exc -> toString exc >>= fail)
                                            (maybe (return "null") toString)
```

With these additions it is suddenly possible to do the following with every function that returns a Java object:

```haskell
runJava $ do
    (AsString string) <- currentThread
    io $ do
        putStrLn "Voila, a String:"
        putStrLn string
```

5.3.2 coerce

The implementation of the much discussed `coerce` function is part of a type class `InstanceOf`. The j2hs tool will create place holders for every type, for example

```haskell
data String'' = String
```

This can be used then to use an `instanceOf` function with the type `JavaObject o => o -> a -> Java Bool` like so:

```haskell
isInstanceOf <- anyType `instanceOf` String
```

This is achieved using an associated type:

```haskell
class InstanceOf a where
    type CoercedType a

    -- | Check if the object of type @a@ is an instance
    -- of the type represented by @b@.
    instanceOf :: JavaObject o => o -> a -> Java Bool

    -- | Check if the object of type @a@ is an instance
    -- of the type @c@, represented by @b@. If so, it will coerce
    -- the object of type @a@ and pass it to the given action.
    --
    -- If @a@ was an instance of @c@ (where @c@ is represented
    -- by @b@) this function will return @'Just' d@, where @d@ is
    -- the result of the optional computation. If not, 'Nothing'
    -- is returned.
    whenInstanceOf :: JavaObject o => o -> a -> (CoercedType a -> Java d) -> Java (Maybe d)

    -- | Coerces the given object of type @a@ to an object of
    -- @c@, where @c@ is represented by a value of type @b@.
    -- Returns @'Nothing'@ if this is not possible.
    coerce :: JavaObject o => o -> a -> Java (Maybe (CoercedType a))

    instanceOf o t =
        whenInstanceOf o t (return . const ())
            >>= return . maybe False (const True)

    whenInstanceOf o t a =
        coerce o t >>= maybe (return Nothing) (fmap Just . a)

    coerce o t = whenInstanceOf o t return
```

5.3.3 Casts

Since the low level interface as well as the mediuem level interface only returns `JObjects` we need a way to create a specific `newtype` from `JObjects` when returning an Object value from a function. This would not be a problem if the type declarations and function declarations were in the same module, but they are not. We also do not want to expose the constructor of `newtype` declarations, as to not allow the encapsulation of illegal references in for example a `String'`.

Thus the j2hs tools specifies a type clas `UnsafeCast` which is for internal usage only. In addition it generates the tools creates an instance of this class for every type:

```

-- | For INTERNAL use only. Is however not in a hidden module,
-- so that other libraries can link against it.
class UnsafeCast a where
    -- | For INTERNAL use only. Do not use yourself.
    unsafeFromJObject :: JObject -> Java a
```

Since the module `Foreign.Java.Bindings` should only be used by code generated by the `j2hs` tool, and never be imported directly by an end users application, this should not pose a problem.

5.4 Summary

The j2hs tool is capable of translating complete libraries, most prominently the Java SE 7 standard library which comprises more than 3000 classes, into Haskell. The resulting Haskell API is very feature rich and offers the programmer the possibility of extending the feature set without actually touching the bindings themselves. This is of course aided by the Haskell type system and certain extensions that have been used in the creation of these Haskell APIs.

6 Example: A Swing Calculator written in Haskell

```haskell
import Foreign.Java
import Control.Monad
import Data.IORef
import Java.Awt.Event
import Javax.Swing
import Javax.Swing.JButton (new'JButton', addActionListener)
import Javax.Swing.GridLayout (new'GridLayout'')
import Javax.Swing.JFrame (new'JFrame, add, setLayout, setTitle)

main = runJavaGui $ do

    currentValue <- newIORef ( 0 ::  Double )
    stack        <- newIORef ([] :: [Double])

    (Just jFrame) <- new'JFrame

    new'GridLayout'' 4 4 >>= setLayout jFrame

    let buttonText = [ "1", "2", "3", "*"
                     , "4", "5", "6", "-"
                     , "7", "8", "9", "+"
                  , "Push", "0", "",  "/" ]

    buttons <- mapM new'JButton' buttonText :: Java [Maybe JButton]

    let updateDisplay = do
            val <- io $ readIORef currentValue
            setTitle jFrame (show val) :: Java ()

        exec = \operation -> do
            xs <- io $ readIORef stack
            case xs of
                [] -> setTitle jFrame "Empty␣Stack!"
                (x:xs) -> do io $ do updateIORef' stack tail
                                     updateIORef' currentValue (flip operation)
                             updateDisplay

        handleAction :: String -> ActionEvent' -> Java ()
        handleAction = \whatFor actionEvent -> do
            case whatFor of
                "" -> return ()
                "Push" -> io $ do
                    v <- readIORef currentValue
                    modifyIORef' stack (v:)
                "*" -> exec (*)
                "-" -> exec (-)
                "+" -> exec (+)
                "/" -> exec (/)
                digit -> do
                    io $ modifyIORef currentValue (\x -> x * 10 + read digit)
                    updateDisplay

    forM (zip buttons buttonText) $ \((Just btn), text) -> do
        () <- jFrame `add` btn
        () <- btn `addActionListener` (handleAction text)

    () <- setSize jFrame 400 300
    () <- jFrame `setLocation` Nothing
    () <- jFrame `setVisible` True
    return ()
```

This is a very simple editor that is written in Haskell using the bindings generated by `j2hs` for Java SE 7.

A short walk-through Lines 1 - 8 contain `import` statements. These tend to become quite large when using functions from many modules. To avoid ambiguity we only import the functions that are really needed.

In line 10 the JVM is started and (only relevant for OS X) any special precautions for running a GUI are taken.

In line 12 and 13 two mutable variables are being set up which hold a *current value* and a stack of values.

In line 15 a new Window is creates (a `JFrame`) which will be displayed in lie 59. We know that the creation of a JFrame can not fail, thus we are being brave and match it directly with `(Just jFrame)` such that `jFrame` contains now a value of type `Javax.Swing.JFrame'`.

In line 17 the set the layout for the frame (actually for the panel inside the frame, but this thesis is not a beginners Java course).

In lines 19 - 24 twelve buttons are created. References to these buttons are being held in `buttons` such that we can add *action listeners* later on.

Line 26 - 28 define an @updateDisplay@ function which reads the currentValue and displays it as the title of the window (we are being modest here and do not add an extra text field for showing the results).

Line 30 - 36 defines a functions which executes a given operation on the head of the stack and the current value. The type of this function is
`(Double -> Double -> Double) -> Java ()`. This function will also store the result in the current value and call `updateDisplay`.

Line 38 - 51 define an action handler, actually several action handlers. Based on the first argument the action handler decides what it will do with the given action event. Sections (partially applied functions) of this will later on act as the real `ActionHandler` objects that we add to our buttons. The most complex operation here is entering a digit and updating the display.

Line 53 - 55 adds the buttons to the `jFrame` and adds the action listeners.

The last three lines adjust the window to a size of 400 x 300 px and center it on the screen (passing `null`, a.k.a. `Nothing`, to `setLocation` causes a Swing JFrame to be centered on the screen).

Note the funny looking invocations of the functions returning void. Since we could equally well decide to get an `Either JThrowable ()` we have to make it explicit that what we want is void. Hence the pattern matching with `()`.

7 Summary

In this thesis a successful translation of Java APIs into Haskell APIs has been conducted. The use of sophisticated features of Haskell's type system allows for a flexible and extensible translation of Java APIs. Also a library which glues Haskell and Java together has been presented Furthermore: A tool that automatically creates high level bindings has been built, and it is capable of translating the API of the Java standard library in such a way that its functionality can be exploited by Haskell application. This was demonstrated by a very simple calculator.

What is missing though is the possibility to call arbitrary Haskell functions from within the Java virtual machine. This is difficult for several reasons: While the FFI absolutely allows for the bidirectional interaction with other languages, there is no simple way of analyzing existing Haskell modules other than analyzing the sources code. It it also not possible to create a library such as the java-bridge which does not need to generate any glue code at all. This is due to the fact that Haskell functions need to be exported explicitly, which again requires us to generate glue code for which we would need to reflect on the Haskell modules that we want to utilize.

Besides these technical problems there are other issues regarding the translation of Haskell APIs to Java: The type system. While at its core a simple language, the type system is partly too complex to be adequately mapped into Java. Namely higher kinded types and subtyping in the same place are very hard to get right (which is presumably the reason for why higher kinded type variables do not exist in Java in the first place).

There is a third argument not properly spelled out yet: It is highly unlikely that the translation of Haskell APIs into Java is useful at all. Many Haskell libraries are fairly general and aim for supporting the Haskell programmer with Haskell idioms that reduce boiler plate or exhibit useful properties which are captured by their types. A good example is the Parsec library which provides parser combinators. Using those in Java would look really awful, let alone since we would need to find a unique representation of operators with only alphanumeric characters.

In the introduction it is said that it might be useful for a Java programmer to outsource certain parts of a Java application to a Haskell application. This is not the same as using Haskell libraries. It is also possible with the java-bridge library, though the programmer has to undertake some manual effort. By means of the `implementInstaceBy` function she can implement a Java interface in Haskell which can transparently be used in Java. All that is left to the programmer is create an exported function that returns the proxy instance for the desired interface. Accessing this native function can be done for example using *Java Native Access (JNA)* a library providing for easier incorporation of foreign code into Java code than the JNI does.

From the perspective of a Haskell program this approach would look like just having a weird entry point, instead of `main :: IO ()` there would be a`main :: InterfaceFunc`.

7.1 Related work

Several other attempts at connecting Haskell and Java have been made, too much to discuss them all. Most of them share a common property: They are defunct by now. Many more look like abandoned hobbyists projects that never had a feature set comparable to the java-bridge and the bindings generator presented in this thesis.

One of the most promising attempts has been made with LAMBADA - HASKELL AS A BETTER JAVA by Erik Meijer and Sigbjorn Finne, published in 2001 in the *Electronic Notes in Theoretical Computer Science 41 No 1.*

Judging by the abstract of the paper it seems to have the same goals as this thesis. Looking in to it we see similar ideas, but there are notable differences. Subtyping for example is not encoded via type classes but via *phantom types*. Also there is no monad that wraps the virtual machine, but the reference to a JVM is hidden away using unsafe operations and IORefs. Calling Haskell code back from Java is completely different, as it uses another tool – "HDiet" – which is capable of loading Haskell libraries as if they were ordinary dynamic libraries and can therefor access arbitrary Haskell functions. "HDiet" seems to be a product on some earlier work which connected COM[40] with Haskell.

The tools presented in the paper are nevertheless also defunct by now. Since it was published in 2001 I assume that it uses an early version of the Haskell FFI. The FFI has only been incorporated into the Haskell language standard in 2011 with the release of Haskell2010.

Another recent development is *Frege*, a Haskell fork that runs in the virtual machine. Frege is not source compatible with Haskell, but it is an ongoing effort to create a functional programming language much like Haskell for the JVM.

7.2 Further directions

Since the feature set of the bindings generated by `j2hs` can be easily extended by defining new instances of the relevant type classes the thought comes to mind to write more marshallers, for example an instance that creates lists from arrays or association lists from maps and so on.

Furthermore it would be nice if the j2hs tool would be able to make use of Java annotations such as `@NotNull`. This way the translated bindings could be more concise and allow for less illegal usage to remain unseen by the type checker.

While I firmly believe that the translation of Haskell APIs to Java in order to use Haskell libraries in Java is useless, I do think that it could be worthwhile to transpile Haskell into Java. Also it might be interesting to add JNI calls to the the Haskell FFI – an option which is explicitly anticipated in the language report: "Only the semantics of the calling conventions ccall and stdcall are defined herein; more calling conventions may be added in future versions of Haskell."[41].

Last but not least: The library needs testing an real world use. At the time of this writing the java-bridge is roughly two months old.

7.3 Where can the java-bridge be found?

The java-bridge and associated tools can be found on HackageDB at `http://hackage.haskell.org/`. It compromises the packages `j2hs`, `java-reflect`, `java-bridge-extras`, and `java-bridge-examples`. You can also do a `cabal install java-bridge` to install the latest version from Hackage (after doing a `cabal update`).

7.4 Acknowledgements

I would like to thank Christian Hörbelt, Konrad Reiche, Benedict Roeser, Georg Schütze and André Zoufahl for proof-reading this thesis. Albeit some of them do neither speak Haskell nor Java, all of them provided me with valuable input.

Also I have to mention how great a place the Haskell community is. During the work on this thesis I created several libraries, like the `multimap` package, that I needed in this project. I have never gotten responses to a software package so fast and with so much support. I will definitely submit an article about the java-bridge in the next Haskell Community Activities and Libraries report.

[40]Component object modell
[41]Haskell 2010 Language Report § 8.4.1 Calling Conventions

A References

THE HASKELL 2010 COMMITTEE, SIMON MARLOW (EDITOR)
The Haskell 2010 Language Report
`http://www.haskell.org/definition/haskell2010.pdf`

JAMES GOSLING, BILL JOY, GUY STEELE, GILLAD BRACHA, ALEX BUCKLEY
The Java Language Specification – Java SE 7 Edition (2013-02-28)
`http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`

SHENG LIANG
The Java Native Interface – Programmer's Guide and Specification
Addison-Wesley, 1999

SIMON MARLOW, SIMON PEYTON JONES, WOLFANG THALLER
Extending the Haskell Foreign Function Interface with Concurrency
Microsoft Research Ltd., Cambridge U.K
Submitted to The Haskell Workshop, 2004
`http://community.haskell.org/~simonmar/papers/conc-ffi.pdf`

BRENT A YORGEY, STEPHANIE WEIRICH, JULIEN CRETIN, SIMON PEYTON JONES, DIMITRIOS VYTINIOTIS
Giving Haskell a Promotion
`http://research.microsoft.com/en-us/people/dimitris/fc-kind-poly.pdf`

THE GHC TEAM
The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.2
`http://www.haskell.org/ghc/docs/7.6.2/html/users_guide/`

SIMON MARLOW
Parallel and Concurrent Programming in Haskell
`http://community.haskell.org/~simonmar/par-tutorial.pdf`

GILAD BRACHA, MARTIN ODERSKY, DAVID STOUTAMIRE, PHILIP WADLER
GJ: Extending the Java programming language with type parameters `http://homepages.inf.ed.ac.uk/wadler/gj/Documents/gj-tutorial.pdf`

SIMON PEYTON JONES
Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell
April 7, 2010
`http://research.microsoft.com/en-us/um/people/simonpj/papers/marktoberdorf/mark.pdf`

SIMON PEYTON JONES, ANDREW GORDON, SIGBJORN FINNE
Concurrent Haskell

List of Figures

1 The polymorphic id function 8
2 A polymorphic List 8
3 A polymorphic method in Java 8
4 This fictitious operator is polymorphic, but 'a' is restricted. 9
5 An instance declaration in Haskell. 9
6 Polymorphic id in Java 9
7 A polymorphic list in Java. 9
8 A polymorphic map in Java 9
9 The same in Haskell 9
10 A higher kinded type variable in Haskell: $c :: * \to *$ 9
11 This is not possible in Java: E is applied to a type variable. 10
12 Non-strict reduction 11
13 Strict reduction 11
14 The calculation from above evaluated inside the Java Virtual Machine. 11
15 Success. 11
16 fail! 11
17 fail! 11
18 Here, non-strict reduction yields something useful from an infinite list 12
19 A Java method declares that it throws an IOException. 12
20 The Java exception hierarchy 13
21 A possible overloading scheme using flexible instances. 16
22 Composability screwed up. 17
23 The Object.toString() method as a Haskell function. 20
24 A Java method as polymorphic Haskell function. 20
25 Translation of a non-static method involving type variables. 21
26 The InstanceOfAType relation realizes subtyping. 21
27 21
28 This results in an ArrayStoreException. 22
29 A typeclass for Java arrays. 23
30 A method void objectArrayMethod(java.lang.String[][]). 23
31 A perfectly valid Haskell representation of an object that implements java.lang.Runnable. 25
32 Representation of a more complex interface: java.lang.Iterable 25
33 An implementation of the empty iterator. 25
34 An example interface and its record which can be used as a callback. . 26
35 A (shortened) exemplary translation of certain Java types. 27
36 An algebraic datatype in Haskell. 28
37 A tagged union in C. 29
38 Translation of the algebraic datatype Maybe to Java. 30
39 The Haskell type class "Functor" as an interface in Java. 30
40 Basic data flow between Haskell and Java 34

41 The interaction of components in the java-bridge library. Green components are written in Haskell, yellow ones in C, blue ones in Java. Black arrows are calls from Haskell into the JVM, read arrows are calls from the JVM back into Haskell. 34

42 C code that converts a local reference to a global reference. This is actually not a function in the java bridge library but a macro vmi. `vmi_t` is the virtual machine interface that is maintained by the library. 36

43 Exemplary usage of `runJava` 43

44 The MethodDescriptor datatype. 44

45 Dependencies between Java packages. 57

46 A sample of the code generated by j2hs: equals(Object), getClass(), and toString(). 59