Purely Functional Data Analysis with Frege and Apache Spark

Master Thesis

September 7th, 2019

Author: Damian Keller
Master of Science in Engineering FHNW

Advisor: Prof. Dierk König
Institute of Mobile and Distributed Systems

Fachhochschule Nordwestschweiz
Hochschule für Technik

Abstract

This thesis covers Frege as a functional programming language in the JVM, the Apache Spark Big Data engine and the possibility of connecting these two technologies. The thesis explains how Frege and Apache Spark can interoperate from a conceptual standpoint and what issues arise when trying to integrate their core concepts. It gives further insights into the obstacles and learnings encountered during development on the integration between Frege and Apache Spark and documents the decision process and the approaches evaluated and chosen as a possible integration between Frege and Apache Spark.
Among the outputs of this project are the Native Declarations necessary to use the basic functionality of Apache Spark in Frege as well as multiple Apache Spark applications written in Frege that show different levels and ways of integration and interoperability between Frege and Apache Spark and elaborates on the properties of such a solution. This includes a sample Frege application calling Java functions underneath through Native Declarations to prove interoperability and an integrated example that executes Frege functions on distributed Apache Spark nodes.

The code produced in this project is open source and can be found on Github:
https://github.com/elrocqe/frege_spark

Declaration of Authorship

I, Damian Keller, declare that this thesis and the work presented in it is my own, original work. All the sources I consulted and cited are clearly attributed. I have acknowledged all main sources of help.

Winterthur, September 7th, 2019

Damian Keller

Table of Contents

Introduction

The core ideas of this thesis evolve around the two topics Big Data and Functional Programming. With the prevalence of more and more data, the field of Big Data grew strongly in recent years, as the desire to learn more about the structures hidden in the data drives further innovation in this area.
Calling itself a unified analytics engine for large scale data processing, Apache Spark is a commonly used framework when working in the context of Big Data. It is written in Scala and Java, and among other languages provides APIs (application programming interfaces) for these languages. Since Apache Spark belongs to the Java ecosystem, applications will be run in the JVM, the Java Virtual Machine.

Functional programming is a programming paradigm that is strongly rooted in the field of mathematics and has been around for many years. With its strong academic background, Haskell is the pioneer language for functional programming, and many other languages such as Java, Scala and Javascript derived principles for their functional programing functionalities from Haskell.
While other languages have adopted more and more principles into their languages, there have also been different attempts to create a programming language that is close to Haskell and works on similar principles such as a language called Frege.
Frege promotes itself as a Haskell in the JVM and tries to bring the properties of functional programming into the JVM. Frege compiles to and strongly interoperates with Java which results in the possibility to work with many different JVM-frameworks such as in this case Apache Spark.
Using a functional programming language in a big data context could provide certain benefits as calculations on large amounts of data usually involve distributing the data as well as the computations onto multiple nodes. In these settings, the pure and side-effect-free nature of functional programming could provide additional stability and predictability. The fact that these side effects will be stated in the type definitions will lead to compile time errors and help reducing issues at runtime.

This thesis aims to look into the intersection between the two topics of Frege and Apache Spark. One of the goals is to investigate whether it is possible to combine these two approaches and what the implications thereof would be. As Apache Spark provides a Java API, and Frege interoperates strongly with Java, it should be possible to connect them straightforward. This thesis provides the necessary guidance on how to use Frege to write an application for Apache Spark as a tool to work in the context of Big Data.
The intention to do this lies on the assumption that using functional programming principles could provide benefits in a distributed Big Data oriented setting. Effects and properties of functional programming languages such as Haskell and Frege should be made available also for use in the JVM and in the context of Apache Spark. Possible benefits would be the type safety that Haskell and Frege provide and the side-effect free nature of these languages, or at least the ability to express side-effects in the type system.

For the integration of these two technologies, a look into some core concepts of Frege and Apache Spark are necessary. This will lay the groundwork for further integrating and being able to run a Frege application on Apache Spark.

Frege

Frege as a Haskell for the JVM

Frege is a functional programming language that is closely modeled after Haskell and runs in the Java Virtual Machine (JVM).

> Like any Haskell, it is purely functional, enjoys a strong static type system with global type inference and non-strict - also known as *lazy* - evaluation. [1]

As noted in one of the core definitions of Frege, the language shares the typical properties of a functional programming language. Functional programming has advantages that could be very useful in the context of Big Data. The main focus of this thesis will lie on the concept of purity of functions.

> A function is pure if it fulfills the following properties: [2]
> - Its return value is the same for the same arguments
> - Its evaluation has no side effects

Pure functions have no side effects, which means they will not have or depend on any hidden state, alter any state or depend on I/O(input/output). Such a behavior can be beneficial when running applications in parallel or concurrently.

In functional programming languages such as Haskell and Frege, effectful computations are separated from pure computations in ways that preserve the predictability and safety of function evaluation. [3]
Therefore, it is possible to distinguish between Frege code that has side effects and pure functions through the type definitions.

A second crucial core property of Frege is also stated in the Frege language documentation.

> Frege compiles to Java, runs on the JVM, and uses any Java library you want. It can be used inside any Java project. [1]

This already indicates the possible interoperability with Java, and therefore interoperability with other JVM libraries and platforms such as in the scope of this thesis Apache Spark.

How Frege works

As mentions above, Frege-Code compiles to Java and is then processed to Java Bytecode and executed in the JVM. For every .fr-file that is written, the compiler creates corresponding .java and .class files which can be found in the output or target folder of your compiling tool.

```
module examples.frege.HelloWorld where

main :: [String] -> IO ()
main args = do
  result = 1.0 + 1.0
  println "Hello World"
  println result
```

snippets.HelloWorld.fr

```
…
final public static Func.U<RealWorld, Short> $main(final
Lazy<PreludeBase.TList<String/*<Character>*/>> arg$1) {
  return PreludeMonad.IMonad_ST.<RealWorld, Short, Short>$gt$gt(
            Prelude.<String/*<Character>*/>println(PreludeText.IShow_String.it,
"Hello World"),
            Thunk.<Func.U<RealWorld, Short>>shared(
                  (Lazy<Func.U<RealWorld, Short>>)(() ->
Prelude.<Double>println(PreludeText.IShow_Double.it, 1.0 + 1.0D))
                )
          );
}
…
```

snippets.HelloWorld.java (compiled output of snippets.HelloWorld.fr)

For the purpose of interoperating Frege with Apache Spark, a deeper look is needed into a few core concepts of Frege.

Native declarations

> A native declaration introduces a new variable v with type t, that will behave like any other Frege variable of that type but is implemented in Java. [4]

Frege and Java can interoperate with each other by providing variables through native declarations that execute Java code in the background. Let's have a look at a short example:

```
main :: [String] -> IO ()
main args = do
  result = 1.0 + 1.0
  println result
  println "Hello World"
```

snippets.HelloWorld.fr

A similar method written in Java:

```
public class HelloWorld {
  public static Double runJava(Double value) {
    System.out.println("Hello World From Java");
    System.out.println("value = " + value);
    return value + 1.0;
  }
}
```

snippets.JavaHelloWorld.java

The native declaration would look like this:

```
native runJava  HelloWorld.runJava :: (Double) -> IO (Double)
```

snippets.JavaHelloWorldIntegration.fr

The native keyword indicates that this is a native declaration, the runJava is the name by which it can be referenced in Frege. HelloWorld.runJava is the corresponding Java-code to which it will be linked, where HelloWorld is the class name and runJava the method name in that class. From the type declaration it is clear that this function takes a Double value as parameter and return a Double, but also might produce side effects (in this case IO), which is the reason why the return value is wrapped in the IO monad.

Native Module

Furthermore, it is possible to write Java directly in a Frege module. To do so, a native module has to be written inside the Frege module as follows:

```
module examples.frege.NativeHelloWorld where
 native module where {
    public static class NativeJavaClass {
    public static Double runJava(Double value) {
      System.out.println("Hello World From Java");
      System.out.println("value = " + value);
      return value + 1.0;
    }
  }
}
```

snippets.NativeHelloWorld.fr

The native declaration can then be used in Frege code to execute the corresponding Java code, even if it is written inside a .fr-file.

```
native runJava  NativeHelloWorld.NativeJavaClass.runJava :: (Double) -> IO (Double)

main :: [String] -> IO ()
main args = do
  result = 1.0 + 1.0
  println "Hello World"
  updatedValue <-runJava result
  println updatedValue
```

snippets.NativeHelloWorld.fr

Notice that the native declaration of the runJava method references the NativeJavaClass inside the native module now.

Mapping Frege functions in Java

As the name already implies, functions are fundamental to any functional programming language. As an example in Frege, please take a look at this function that takes a double value as input parameter x and returns the incremented value:

```
addOne :: Double -> Double
addOne x = x + 1.0
```

snippets.Tryout.fr

The compiled Java code of such simple methods look very straightforward, as follows:

```
final public static double addOne(final double arg$1) {
  return arg$1 + 1.0D;
}
```

snippets.Tryout.java (compiled output of snippets.Tryout.fr)

But a more complex function, for example the same functionality as above, but written in point-free style will look like this in Frege:

```
addOnePF :: Double -> Double
addOnePF = (+ 1.0)
```

snippets.Tryout.fr

PointFree style is a common pattern to write functions as a composition of other functions and is also called tacit programming. The style is not relevant at the moment, but it forces a difference in how the function is compiled, as the corresponding compiled Java code would now look as follows:

```
final public static Lazy<Double> addOnePF(final Lazy<Double> arg$1) {
  return PreludeBase.<Double, Double, Double>flip(
            (Func.U<Double, Func.U<Double, Double>>)
((final Lazy<Double> n$7656) -> (Func.U<Double, Double>)
((final Lazy<Double> n$7657) -> Thunk.<Double>shared(
    (Lazy<Double>)(() -> (double)n$7656.call() + (double)n$7657.call())
    ))),
    Thunk.<Double>lazy(1.0D), arg$1
  );
}
```

snippets.Tryout.java (compiled output of snippets.Tryout.fr)

To understand how Functions in Frege are mapped to Java, a closer look at the Frege Runtime System and the concepts of lazy evaluation and partial application is necessary.

Lazy Evaluation

Evaluation of an expressions basically means reducing the expression as much as possible, until it cannot be further reduced. This then is the simplest form and the resulting expression is irreducible, usually called a value.

Haskell uses the concept of lazy evaluation, or non-strict evaluation, and Frege shares the same properties. Lazy evaluation means that expressions are not evaluated until it is absolutely needed by other terms referring to them.

In Frege, this is implemented through the functional interface Lazy in the package frege.run8.

```
package frege.run8;

import java.util.concurrent.Callable;

/**
 * Common interface of all lazy values.
 *
 * @author ingo
 *
 */
// @FunctionalInterface
public interface Lazy<R> extends Callable<R> {

        /**
         * <p> Compute the value if it is needed. </p>
         *
         * @see java.util.concurrent.Callable#call()
         */
        public abstract R call();

        /**
         * <p> Tell if this is really a {@link Thunk} </p>
         * @return
         */
        public default Thunk<R> asThunk() { return null; }
```

```
        /**
         * <p> Tell if this is shared. </p>
         * <p> Data and functions whose {@link Lazy#call()} method returns <b>this</b> as well
         * as simple boxes that just hold a value ready to be supplied and {@link Thunk}s
         * are considered shared.
         * <p> But a bare lambda expression is assumed to be in need of sharing. For example:
</p>
         * <code> () -&gt; 42 </code>
         * @return false, if sharing this would make any sense, otherwise true
         */
        public default boolean isShared() { return false; }

}
```

Frege.run8.Lazy.java

As stated in the interface definition, the Lazy-interface imports and extends the Callable-interface of java.util.concurrent.

> public interface **Callable<V>**
> A task that returns a result and may throw an exception. Implementors define a single method with no arguments called call.
>
> The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception. [5]

Frege.run8 therefore inherits the method call() from the Callable-interfac and therefore the value of the Lazy-object will not be executed unless the call() method is called. This represents the mechanics to defer evaluation of an expression, which is how Frege maps this concept to its Java compilation.

Partial Application

Partial Application is strongly coupled with the concept of currying.

> Currying is when a function that takes multiple arguments is broken down into a series of functions that take part of the arguments. [6]

or from a different perspective

> Partial application involves passing less than the full number of arguments to a function that takes multiple arguments. [7]

Take a look at the following simple example, where the addOne-function partially applies the add function by passing it one of its parameters but deferring the other one.

```
add :: Double -> Double -> Double
add x y = x + y

addOne :: Double -> Double
addOne y = add 1.0 y
```

snippets.Tryout.fr

Lazy Evaluation and Partial Application in Frege

Frege implements these functional programming behaviors through its runtime system. The following is stated in the Frege Wiki:

> The main concern of the run time system is to provide for lazy evaluation and partial function application.
>
> The Frege Run Time System consists of a handful of interfaces and abstract classes, which are used to implement laziness, function types, and higher kinded types [8]

Since functions in Frege are lazy, this behavior has to be mapped when compiling Frege-Code to Java, which is done through the aforementioned Lazy interface.

Frege offers the interface frege.run8.Func.U to compile a function.

```
…
public class Func  {
  @FunctionalInterface public interface U<𝒜, ℬ>
    extends Lazy<Func.U<𝒜, ℬ>>, Kind.U<Func.U<𝒜, ?>, ℬ>, Kind.B<Func.U<?, ?>, 𝒜, ℬ>
   {
    public Lazy<ℬ> apply(final Lazy<𝒜> a) ;
    public default Func.U<𝒜, ℬ> call() {
      return this;
    }
    public default boolean isShared() {
      return true;
    }
  }
…
}
```

frege.run8.Func.java

The class Func also introduces functional interfaces for different amount of type parameters (here <𝒜, ℬ, 𝒞…>) to represent various Frege functions.

```
…
  @FunctionalInterface public interface B<𝒜, ℬ, 𝒞>
    extends Lazy<Func.B<𝒜, ℬ, 𝒞>>, Kind.U<Func.B<𝒜, ℬ, ?>, 𝒞>,
      Kind.B<Func.B<𝒜, ?, ?>, ℬ, 𝒞>, Kind.T<Func.B<?, ?, ?>, 𝒜, ℬ, 𝒞>
   {
    public Lazy<𝒞> apply(final Lazy<𝒜> a, final Lazy<ℬ> b) ;
    public default Func.B<𝒜, ℬ, 𝒞> call() {
      return this;
    }
    public default boolean isShared() {
      return true;
    }
  }
…
```

frege.run8.Func.java

Whereas Func.U takes one parameter and return one result value, Func.B will take two values and return one result value. This represents the way in which Frege maps partial application to a single function-object in the JVM.
All the corresponding interfaces can be found in the class frege.run8.Func.java

How a Frege function is compiled to Java

To have an in-depth look at all the parts involved, please have another look at the function

```
addOnePF :: Double -> Double
addOnePF = (+ 1.0)
```

snippets.Tryout.fr

and the generated Java output

```
final public static Lazy<Double> addOnePF(final Lazy<Double> arg$1) {

  return PreludeBase.<Double, Double, Double>flip(

            (Func.U<Double, Func.U<Double, Double>>)
((final Lazy<Double> η$7656) -> (Func.U<Double, Double>)
((final Lazy<Double> η$7657) -> Thunk.<Double>shared(
                          (Lazy<Double>)(() -> (double)η$7656.call() +
(double)η$7657
                              .call())
                        ))),
            Thunk.<Double>lazy(1.0D), arg$1
          );
}
```

snippets.Tryout.java (compiled output of snippets.Tryout.fr)

addOnePF here is a Java method that takes an argument arg$1 of type Lazy<Double> and returns a result of type Lazy<Double>.

The definition of PreludeBase.flip is: flip f a b = f b a
flip just flips the arguments of a function, where in this example Thunk.<Double>lazy(1.0D) would be the first value argument a of flip, and the incoming method parameter of addOnePF, arg$1, would be the second value parameter b.
This leaves the junk in the middle to be the function f.
The first line indicates the partial application, since those are nested functions.
Func.U<Double, Double>>) is a function that takes a Double and returns a Double.
Func.U<Double, Func.U<Double, Double>>) is a function that takes a Double and returns another function that takes another Double and return a Double value as result.

η$7656 and η$7657 indicate the incoming parameters of this functions. There are some casts involved, but the crucial part is the call()-methods on the parameters. The Lazy<Double> values force the values to be evaluated to compute the result of the two values, and a new Lazy<Double> is returned with the result.
The -> indicate an anonymous function in Java, and here this means: for an incoming value η$7656, return a Func.U that for an incoming value η$7657 performs the addition on the lazy values.

The knowledge that Frege functions are mapped to Java through the interface Func.U and the other interfaces defined in class frege.run8.Func and that lazy evaluation is implemented through the Lazy-interface will be useful later on when executing Frege Functions in Apache Spark.

Apache Spark

To prove the interoperability with Frege and Apache Spark, it should be shown that an Apache Spark program can be written in Frege and provide the same functionality as the versions in other languages. To understand the integration from Apache Spark's side, a closer look at some core concepts are needed.

The Three Apache Spark APIs - RDD, DataFrame and DataSet

Apache Spark provides three different data abstraction APIs, namely RDD, DataFrame and DataSet. These APIs serve different purposes and have different properties, which will not be further explained here. This thesis focuses on RDD while it also provides integration for the basic DataFrame functionality, as RDDs are generally used when manipulating data with lambda functions rather than a DSL as it is the case with the DataSet-API.

RDD – Resilient Distributed Dataset

RDDs were the initial data abstraction of Apache Spark and have the following features. [9]

- **Resilient**, i.e. fault-tolerant with the help of RDD lineage graph and so able to recompute missing or damaged partitions due to node failures.
- **Distributed** with data residing on multiple nodes in a cluster.
- **Dataset** is a collection of partitioned data with primitive values or values of values, e.g. tuples or other objects (that represent records of the data you work with).
- **In-Memory**, i.e. data inside RDD is stored in memory as much (size) and long (time) as possible.
- **Immutable** or **Read-Only**, i.e. it does not change once created and can only be transformed using transformations to new RDDs.
- **Lazy evaluated**, i.e. the data inside RDD is not available or transformed until an action is executed that triggers the execution.
- **Cacheable**, i.e. you can hold all the data in a persistent "storage" like memory (default and the most preferred) or disk (the least preferred due to access speed).
- **Parallel**, i.e. process data in parallel.
- **Typed** — RDD records have types, e.g. Long in RDD[Long] or (Int, String) in RDD[(Int, String)].
- **Partitioned** — records are partitioned (split into logical partitions) and distributed across nodes in a cluster.
- **Location-Stickiness** — RDD can define placement preferences to compute partitions (as close to the records as possible).

It is important to note here that working with RDDs involve some of the same concepts known from functional programming. RDDs are immutable, therefore have no hidden state and they are lazy evaluated which corresponds to the same concepts Frege works with.

Transformations and Actions

When working with RDDs in Apache Spark, data is manipulated with functions, and it is important to keep in mind that Apache Spark distinguishes two different operation types.

Transformation will conceptually transform or alter the data in a RDD. But applying a transformation to an RDD will result in a new RDD with altered data since RDDs in Apache Spark are immutable. Typical transformations are map and filter, or also groupBy and join. Transformation operations will not be performed immediately. To get a non-RDD value, for example an aggregation result, an action has to be called on the RDD. Typical actions are reduce(func), collect(), count() or take(n). The important information to keep in mind here is that transformations will only be applied once an action is executed, which corresponds to Apache Spark's lazy evaluation. [10] [11]

DataFrame and DataSet

The relationship between these two APIs is stated in the documentation as follows:

> In Scala and Java, a DataFrame is represented by a Dataset of `Row`s. In the Scala API, `DataFrame` is simply a type alias of `Dataset[Row]`. While, in Java API, users need to use `Dataset<Row>` to represent a `DataFrame`. [12]

This is why this project will only focus on the DataSet-API as it focuses on working with Frege and therefore Java.
DataSet is a more high-level API that uses a DSL (domain-specific language) and puts focus on structured data. It is generally faster and uses less memory which is why it makes sense to use the DataSet-API when the code optimization, performance and space efficiency are crucial.

For a quick comparison, the following example usages of RDD and DataSet/DataFrame were taken from the Apache Spark Examples:

```
JavaRDD<String> textFile = sc.textFile("...");
JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
```

```
Dataset<Row> df = spark.read().json("examples/src/main/resources/people.json");
df.groupBy("age").count().show();
```

[13]

Since the different APIs have various advantages and disadvantages and are used in different use cases, interoperability should be provided with both main APIs. From a Frege perspective, it doesn't make a difference which API to use. Basic integration to call Apache Spark from Frege was implemented for both APIs and the native declarations for the corresponding classes can be found in /src/main/frege in the packages under bindings.spark. Most work for providing full integration was only done on the RDD-API to proof general feasibility of running Frege functions on Apache Spark.

Functions in Apache Spark

The use of functions in Apache Spark evolves around the Function and Function2-interfaces of the org.apache.spark.api.java.function-package.

```
package org.apache.spark.api.java.function;

import java.io.Serializable;

// doc comment ommitted

@FunctionalInterface
public interface Function<T1, R> extends Serializable {
  R call(T1 v1) throws Exception;
}
```

```
package org.apache.spark.api.java.function;

import java.io.Serializable;

/**
 * A two-argument function that takes arguments of type T1 and T2 and returns an R.
 */
@FunctionalInterface
public interface Function2<T1, T2, R> extends Serializable {
  R call(T1 v1, T2 v2) throws Exception;
}
```

Function takes one argument, for example to manipulate a value, while Function2 takes two arguments, for example to perform an aggregation function like sum.
In Apache Spark, higher-order functions like filter and map will take an object of type Function as an argument, while reduce takes a Function2-object as argument.
In newer versions of Apache Spark, it is also possible to pass lambda expressions as functions, but this functionality was not further investigated.

The basic word count example of Apache Spark shown such usage. [13]

```
JavaRDD<String> textFile = sc.textFile("...");

JavaPairRDD<String, Integer> counts = textFile
    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
    .mapToPair(word -> new Tuple2<>(word, 1))
    .reduceByKey((a, b) -> a + b);
```

For example in the mapToPair, a lambda is passed that creates a tuple for every word and adds the count as second value. The lambda-function could also be written with a Function.

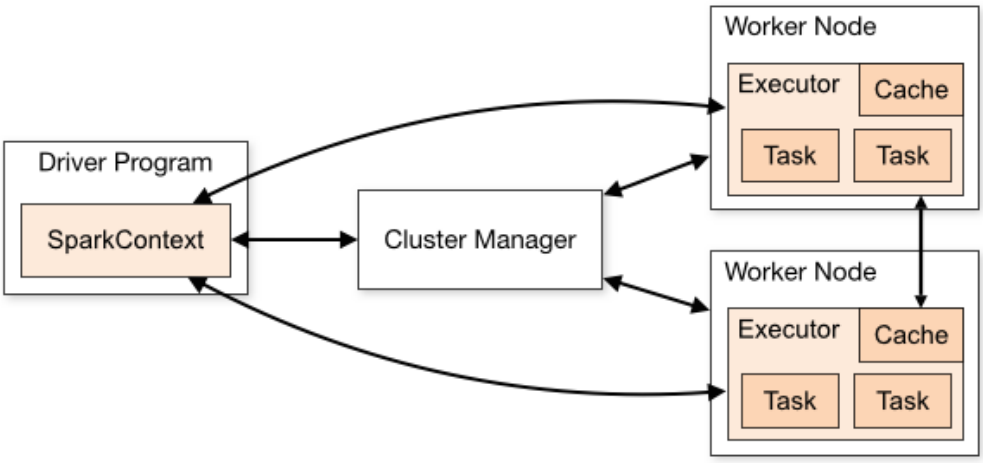
```
new Function<String, Tuple2<>() {
       public Tuple2<> call(String word) {
              return new Tuple2<>(word, 1);
       }
};
```

Please keep in mind that when working with the DataSet-API, there are related Function classes to be used, namely the FilterFunction, MapFunction and ReduceFunction in the org.apache.spark.api.java.function package.

Apache Spark Distributed Execution

There are three main components when running Apache Spark: the cluster manager, the driver and one or multiple worker nodes.

The driver will schedule tasks and the worker nodes will execute them while the cluster manager allocates resources. There are different cluster managers, but for simplicity, the Spark standalone mode was chosen.



[14]

The easiest way to run an Apache Spark application is in local mode. This will run the driver and the worker of Apache Spark in the same Java process and not further distribute the work.

If you want to run a distributed Apache Spark application, it can be run in a cluster and Apache Spark will take care of running the calculations on multiple nodes.

For more information on how to run Apache Spark in the different modes, please refer to the upcoming chapter “Setup”.

Setup

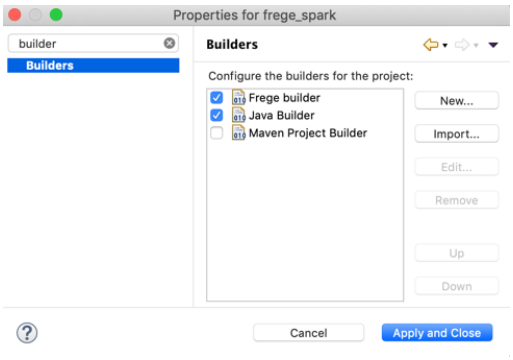
Repository

The sources of this project can be found on Github under
https://github.com/elrocqe/frege_spark

Please read the installation guide in the Appendix A on how to set up the environment.

Tools

Eclipse is currently the best supported IDE to work with Frege as there exists an Eclipse plugin for the language. The corresponding installation guide for Eclipse and its Frege plugin can be found in the Github repository of fregIDE, the frege eclipse plugin:
https://github.com/Frege/eclipse-plugin



After enabling Frege in the project, *.fr-Files will be processed by the Frege-Compiler. Since both Java and Frege source files are used, there are also two builders needed for the project. Make sure in the properties of the current project in Eclipse that the builders are set up properly.

Dependencies

The necessary dependencies such as Apache Spark are added by using Maven for dependency management. The definitions can be found in the pom.xml and the libraries are then linked as referenced libraries.

Output

The *.fr-sources get compiled by the Frege builder while the Java builder processes the *.java files. The output will then be placed in the defined location, in our case under target/classes.
In there are files from the Java sources as well as the Frege sources. For each Frege file (*.fr), there exists a corresponding *.java file with the compiled java output of the given *.fr file. And for each *.java file the corresponding *.class file can be found, which provides the executable Java bytecode generated by the Java compiler.
Please note the following comment of the FregIDE documentation:

> Note that the order of Java and Frege builder is important for polyglot programming: If you have Java source code in the project that you are going to call from Frege, the Java builder should run first (this is the standard case). If, however, the java code calls into Frege, the build order should probably be reversed. The build order can be changed in the project properties. [15]

For further information about the setup and a dedicated the installation guide, please refer to the Appendix.

Deployment

To run an Apache Spark application in cluster mode, the application must be provided as a pre-compiled JAR (Java Archive). Therefore, the output files of this project are packed into a JAR which then is linked and made available to Apache Spark.
This can be done either through the Constructors or the addJar-Method of the JavaSparkContext/SparkContext-Class.

```
JavaSparkContext(String master, String appName, String sparkHome,
String jarFile)

JavaSparkContext(String master, String appName, String sparkHome,
String[] jars)
```

addJar(String path)

Adds a JAR dependency for all tasks to be executed on this SparkContext in the future.

[16]

By providing the JAR file, the code will be available to be executed on the nodes. In this project, the application gets packed into a Java Archive called frege-spark.jar which is added as an executable jar when invoking Apache Spark. For convenience, the latest version of frege-spark.jar can also be found in the source code repository.

For this to work, the dependencies of the project, which include the needed Frege functions in the frege-spark.jar, and the frege-interpreter dependencies need to be passed to the Apache Spark application

```
list <- arrayFromListST [applicationJar, fregecJar, interpreterJar]
sparkConfig.setJars list
```

whereas the exact filenames are fetched from Config.fr.

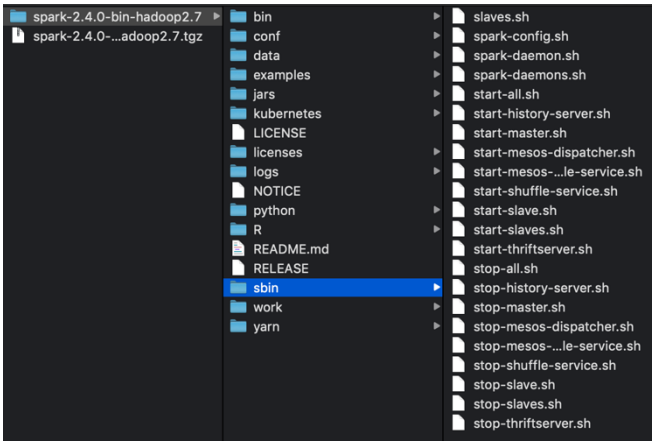
```
module spark.config.Config where

file :: String
file = "data/first.csv"

fregecJar :: String
fregecJar = "frege3.24.405.jar"

applicationJar :: String
applicationJar = "frege-spark.jar"

interpreterJar :: String
interpreterJar = "jars/frege-interpreter-core-1.3-SNAPSHOT.jar"
```

config.Config.fr

Currently, the frege-spark.jar can be created by exporting the module in eclipse into a jar-file. This could be automated into a build command using a build management tool, for example maven.

Local Apache Spark

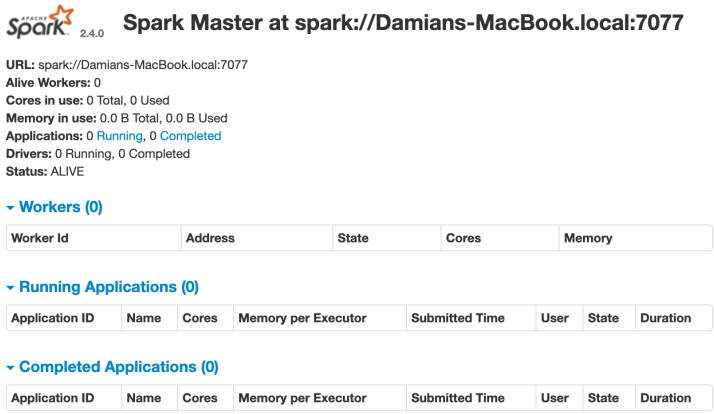
The simplest way to run an Apache Spark application is just to add the dependency through the dependency management tool, in this case Maven, and run the application directly which will spawn an Apache Spark instance.
To run Apache Spark in a distributed cluster locally, a local distribution of Apache Spark is needed. The latest ones can be downloaded from https://spark.apache.org/downloads.html. After unzipping the downloaded file, the following folder structure and files will be present:



To start a distributed cluster, a master and a slave node have to be spawned. This can be done by running the starter scripts in the sbin folder inside a terminal.

./start-master.sh
./start-slave <your-master-url>
e.g. ./start-slave.sh spark://Damians-MacBook.local:7077

Afterwards, the status of the Apache Spark instance can be checked on http://localhost:8080



In this web application, all the active workers, running and completed applications will be shown and you can also access the logs of the applications on the worker nodes.

Adjust the Apache Spark Config to

```
sparkConfig.setMaster "spark://Damians-MacBook.local:7077"
```

instead of

```
sparkConfig.setMaster "local"
```

and the application then will be run on the defined cluster.

Source Files

The source files are separated into Frege source code, which is located under src/main/frege/ and the Java source code in src/main/java/.

src/main/frege	
bindings	bindings to create Apache Spark functions
bindings.custom	custom bindings for examples
bindings.spark	native declarations for Apache Spark classes
bindings.spark.sql	native declarations for Apache Spark classes
bindings.testing	testing bindings for assertEquals
config	configuration file
examples	executable Frege Apache Spark applications
examples.helpers	Frege helper class for examples
functions	Frege FunctionPool to be executed (e.g. via interpreter)
snippets	Frege code snippets
utils	small tool to create example input file

src/main/java	
bindings	corresponding Java code for the bindings
bindings.custom	corresponding Java code for the bindings
examples	executable Java application
functions	Java FunctionPool to be executed (e.g. via reflections)
script	helper class for Frege interpreter
snippets	Java code snippets

Executable Examples

Runnable Apache Spark applications are named …Example.fr or …Example.java and can be found in the examples-package. Code found in the snippets-package may not contain Apache Spark applications, but can also be executed.

src/main/frege
- examples
  - DataSetIntegrationExample.fr
  - FregeRDDExample.fr
  - HelloSparkExample.fr
  - IntegrationRDDExample.fr
  - InteroperabilityRDDExample.fr
  - InterpretationRDDExample.fr
  - IOIntegrationExample.fr
  - NativeModuleRDDExample.fr
  - ReflectionRDDExample.fr

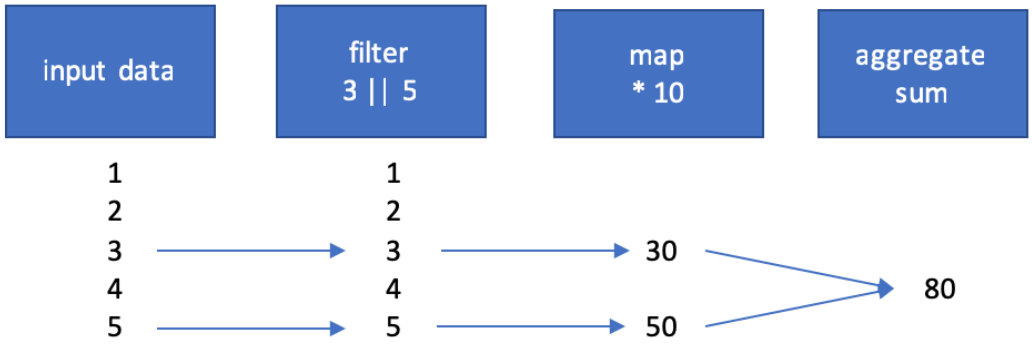
src/main/java
- examples
  - InterpreterExample
  - JavaDataFrameExample.java
  - JavaRDDExample.java

These applications can be started by using dedicated runConfigurations in Eclipse.

Integration

The goal of this project is to write Apache Spark applications in Frege. Such an application should include commonly used transformations like filter, map and an aggregation action like reduce. As mentioned before, Spark will only execute the transformations when an action is called, therefore there must be an aggregation action like sum.

An example application could look as follows:



From a project perspective, it makes sense to divide the task of writing an Apache Spark application in Frege into multiple milestones.

1. **Milestone - Baseline Apache Spark application in a supported language**
   Provide an example application in one of the languages supported by Apache Spark, for this project namely in Java.

2. **Milestone - Frege Application calling Java through native declarations**
   Write native declarations for all needed Apache Spark classes and methods and provide other unknown parts in Java through custom bindings.

3. **Milestone - Use Frege to make calculations on RDD-Data**
   Write Frege Functions to perform transformations on the data. For this, the RDD has to be converted into a Frege collection on which Frege then can execute high-order functions.

4. **Milestone - Pass Frege Functions to Apache Spark to make calculations**
   Write an application that passes Frege functions as parameters to Apache Spark and use those functions to perform transformations on the RDDs.

To get to the intended goal of running an Apache Spark application from Frege, the above milestones were targeted. This allowed to go from a simple, already working example and move more and more towards integrating Frege.

Apache Spark in Java

Different languages are supported by Spark such as Scala, Java, Python and R. Using Java for this initial baseline example makes sense since the Frege code will be compiled to Java in an intermediate step of its compilation. This way, it is possible to compare the Java code and the Java compilation of the Frege code against each other.
While Apache Spark is written in Scala and will eventually be run in the JVM, having a first example in Java provides further benefits, given that the general interoperability between Frege and Java is already guaranteed and it is possible to call Java code from Frege.
From a running Java example, our example can be extended by first calling it from Frege and then further outsource Java code to Frege.

Please have a look at the code of the Java example:

```
package examples;

import org.apache.spark.api.java.*;

public class JavaRDDExample {

    public static void main(String[] args) {
        JavaSparkContext sc = new JavaSparkContext("local", "JavaRDD
Numbers Example");
        JavaRDD<String> rdd = sc.textFile("data/first.csv");

        JavaRDD<Double> resultRdd = rdd
                .map(line -> Double.parseDouble(line))
                .filter(value -> value == 3.0 || value == 5.0)
                .map(x -> x * 10);
        Double result = resultRdd.reduce((a,b) -> a + b);
        sc.close();
        };
}
```

examples.JavaRDDExample.java

A JavaSparkContext is created, the content of a data-file is read and then transformations are executed on the resulting RDD. At the end, there is an action that reduces the values to its sum to be presented as a result of the computations.

An alternative version using the Apache Spark Function types instead of lambdas would look as follows:

```
JavaRDD<Double> resultRdd = rdd
        .map(line -> Double.parseDouble(line))
        .filter(filterThreeOrFive)
        // alternatively
        //.filter(new Function<Double, Boolean>() {
        //    public Boolean call(Double input) {
        //        return input == 3.0 || input == 5.0;
        //    }})
        .map(timesTen);
Double result = resultRdd.reduce(getSum);
```

examples.JavaRDDExample.java

whereas the corresponding Functions are defined as:

```
     public static Function<Double, Boolean> filterThreeOrFive = new
Function<Double, Boolean>() {
     public Boolean call(Double input) {
          return input == 3.0 || input == 5.0;
     }
};

public static Function<Double, Double> timesTen = new Function<Double,
Double>() {
     public Double call(Double input) {
          return input * 10.0;
     }
};

public static Function2<Double, Double, Double> getSum = new
Function2<Double, Double, Double>() {
     public Double call(Double x, Double y) {
          return x + y;
     }
};
```

examples.JavaRDDExample.java

Calling Apache Spark from Frege

To be able to write the same code in Frege, bindings would be needed for all the spark-specific functions such as map and filter. Frege also supports the Java constructs of instance creation which will be needed to set up and run a JavaSparkContext.

The following code illustrates the native declarations that bind Frege to the Java functions, using the example of the class JavaSparkContext.

```
module bindings.spark.JavaSparkContext where

import bindings.spark.JavaRDD
import bindings.spark.SparkConf

data JavaSparkContext = native org.apache.spark.api.java.JavaSparkContext where

    -- constructor
    native new :: Mutable s (SparkConf) -> STMutable s (JavaSparkContext) | String -> String
-> STMutable s (JavaSparkContext)

    native textFile :: Mutable s JavaSparkContext -> String -> ST s (JavaRDD String)

    native master :: Mutable s JavaSparkContext -> ST s String
    native appName :: Mutable s JavaSparkContext -> ST s String
    native version :: Mutable s JavaSparkContext -> ST s String
```

bindings.spark.JavaSparkContext.fr

These bindings correspond to the following constructors and methods from Apache Sparks JavaSparkContext.java.

Constructor Summary

Constructors

Constructor and Description
`SparkContext()` Create a SparkContext that loads settings from system properties (for instance, when launching with ./bin/spark-submit).
`SparkContext(SparkConf config)`
`SparkContext(String master, String appName, SparkConf conf)` Alternative constructor that allows setting common Spark properties directly
`SparkContext(String master, String appName, String sparkHome, scala.collection.Seq<String> jars, scala.collection.Map<String,String> environment)` Alternative constructor that allows setting common Spark properties directly

[16]

```
public RDD<String> textFile(String path)
public String master()
public String appName()
public String version()
```

The Frege compiler will throw errors and indicate, if the corresponding Java methods don't exist or the parameters are not matching.

For each Java class that is used there must exists a corresponding data type in Frege, and for each method used in that class, the data type declaration must have a corresponding native declaration.

For our application using the RDD-API, the following bindings are needed:

```
org.apache.spark.api.java.JavaRDD            ->   spark.bindings.JavaRDD
org.apache.spark.api.java.JavaSparkContext   ->   spark.bindings.JavaSparkContext
org.apache.spark.SparkConf                   ->   spark.bindings.SparkConf
```

The bindings are located in src/main/frege in the package bindings.spark and provide Frege functions to execute the underlying Apache Spark Java methods.

To make the simple application work in Frege, native declarations for the custom function that are passed to the filter, map and reduce functions are also provided.
Here the example of the filter function:

```
data Function a b = pure native "org.apache.spark.api.java.function.Function" where

  pure native filterThreeOrFiveDouble
"bindings.custom.FilterFunctions.filterThreeOrFiveDouble"     :: Function Double Bool
```

bindings.custom.CustomFunction.fr

```
public static Function<Double, Boolean> filterThreeOrFiveOnDouble = new Function<Double,
Boolean>() {
    public Boolean call(Double input) {
        return input == 3.0 || input == 5.0;
    }
};
```

bindings.custom.FilterFunctions.java

The same goes for the mapping and reducing functions and can also be found in the bindings.custom-package. Filter, map and reduce are common high-order functions, and they are defined as pure in the native declarations in bindings.spark.JavaRDD.fr, which will allow to work with them in a side-effect free context. This requires the developer to make sure that

neither the Apache Spark Java code nor the custom filter functions written in Java keep a state or have any other side effects. By adding the pure keyword in the native declaration of our custom filter function, it is guarantee that this function and the code underneath is pure and can be used under that assumption inside the Frege code.

```
  pure native filter        :: JavaRDD a -> Function a Bool -> JavaRDD a
  pure native map           :: JavaRDD a -> Function a b -> JavaRDD b
  pure native reduce        :: JavaRDD a -> Function2 a a a -> a
```

bindings.spark.JavaRDD.fr

The resulting Frege application that spawns a local Apache Spark instance and executes our code looks as follows:

```
sc :: MutableIO JavaSparkContext <- JavaSparkContext.new sparkConfig

-- currentData :: JavaRDD.JavaRDD String
currentData <- JavaSparkContext.textFile sc "data/first.csv"
-- parsedData :: JavaRDD.JavaRDD Double
parsedData = currentData.map  Function.convertToDouble
-- filteredData :: JavaRDD.JavaRDD Double
filteredData = parsedData.filter  Function.filterThreeOrFiveDouble
-- mappedData :: JavaRDD.JavaRDD Double
mappedData = filteredData.map  Function.timesTenDouble
-- sum :: Double
sum = mappedData.reduce Function2.getSum
```

examples.InteroperabilityRDDExample.fr

Computations in Frege

So far, the calculations and transformations are executed through Apache Spark's functions filter, map and reduce. The concept of these high-order functions was initially obtained from functional programming and this approach is now used as a general pattern in other languages as well such as Javascript or, as seen here, in Java. Frege has its own versions of filter and map, while reduce corresponds to what is known as fold in functional programming.

```
filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
foldr :: (a -> b -> b) -> b -> [a] -> b
```

This opens the question whether it is possible to use this functionality in Frege directly to use Frege functions for the calculations.
In our example, the corresponding functions of the Spark Class JavaRDD were used.

```
public JavaRDD<T> filter(Function<T,Boolean> f)
public static <R> JavaRDD<R> map(Function<T,R> f)
public static T reduce(Function2<T,T,T> f)                    [17]
```

Sparks filter, map and reduce methods work on the JavaRDD type, whereas functions in Frege work on the Frege List type, e.g. [String], or [Double], or more generally [T].

The RDD-Class has a method called collect() which return the current RDD as a java.util.list-type.

```
public static java.util.List<T> collect()                     [17]
```

This can be used to retrieve a list of elements from an RDD. There is some conversion needed to get a [String] from a (List String)-data type, but as soon as there exists a Frege-list such as [String], the build-in high-order functions like filter and map can be used to work with the dataset.

The pure Frege functions that are passed to these high-order functions to be executed are defined as follows:

```
-- filter
isThreeOrFive:: Double -> Bool
isThreeOrFive x = (x == 3.0 || x == 5.0)

-- map
timesTen :: Double -> Double
timesTen x = x * 10

-- reduce
getSum :: Double -> Double -> Double
getSum x y = x + y
```

examples.helpers.FregeHelpers.fr

With all the necessary native declarations implemented, it is possible to write Frege code that spawns an Apache Spark application, gets the data collection as an RDD, converts it to a list to let Frege do the transformations and calculations.

```
module examples.FregeRDDExample where

-- imports omitted

main :: IO ()
main = do
  sparkConfig <- SparkConf.new ()
  sparkConfig.setMaster "local"
  sparkConfig.setAppName "Frege-Spark"

  -- sc :: MutableIO JavaSparkContext
  sc :: MutableIO JavaSparkContext <- JavaSparkContext.new sparkConfig

  -- currentData :: JavaRDD.JavaRDD String
  currentData <- JavaSparkContext.textFile sc "data/first.csv"
  -- collectedData :: MutableIO (List String)
  collectedData <- currentData.collect

  -- collectedDataEntry :: Maybe String
  collectedDataEntry <- collectedData.get 0
  print "collectedDataEntry: "
  println $ show $ collectedDataEntry

  -- iterator :: MutableIO (Iterator String)
  iterator <- (collectedData.iterator)
  -- currentList :: [String]
  currentList <- (iterator.toList)
  -- parsedList :: [Double]
  parsedList = map readString currentList
  --filteredList :: [Double]
  filteredList = filter isThreeOrFive parsedList
  -- mappedList :: [Double]
  mappedList = map timesTen filteredList
  -- sum :: Double
  sum = foldr getSum 0.0 mappedList
```

examples.FregeRDDExample.fr

What is executed in the background and what impact it has on the performance needs to be investigated. Apache Spark runs optimizations on transformations which won't be run when the calculations are executed in Frege.

Furthermore, it may not make much sense to use Spark to create an RDD just to convert it back into a List and then let Frege do the calculations, but this proves it is possible nevertheless and the used Frege functions are pure and can later be used when passing Frege functions to Apache Spark.

So far, the following possibilities to run an Apache Spark application were shown:

- Run Java application
- Run Frege application that call methods implemented in Java
- Run Frege application that uses Java bindings to get an RDD, which can be converted to a List in Frege, and use Frege's own high-order function like filter and map

Passing functions

The next step would be to be able to write functions in Frege and pass them as parameters when using Sparks filter and map methods. Let's have a closer look at the involved function definitions:

```
filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
```

A function is passed as an argument to the filter or map function together with the list to be worked on, and a list will be returned as a result.
For filter, the passed function is of type (a -> Bool), which takes an arbitrary value and returns a Bool. This function checks a value of type a and only returns the value if it fulfills a certain condition.
These type definitions already indicate that in Frege, filter and map are pure functions. If filter or map are passed a function and a list without side effects, it will return another list, also without side effects. The compiler will enforce that a function with side effects, e.g. IO (a -> b) cannot be passed and executed.

For comparison, the filter and map functions in Apache Spark have the following method signature:

```
public <T>   JavaRDD<T>   filter(Function<T,Boolean> f)
public <T,R> JavaRDD<T,R> map(Function<T,R> f)
```

A function is given as a parameter as well, so the Function<T,Boolean> f or Function<T,R> f in the Apache Spark method corresponds to (a -> Bool) or (a -> b) in the Frege implementations.

```
filter: Function<T,Boolean>  ->     (a -> Bool)
map:    Function<T,R>        ->     (a -> b)
```

The second parameter in the Frege type definition, the list to operate on, is passed as a second parameter in Frege. But in the Java implementations, this parameter is not needed since the filter and map functions belong to the JavaRDD class on which the functions are invoked. The functions will be called on the current instance of a JavaRDD through the dot-notation.

```
…
JavaRDD currentRDD = …;
Function f = …;
JavaRDD newRDD = currentRDD.filter(f);
…
```

This shows that the different implementations on filter and map strongly correlate between the Frege function and the Java methods used in Apache Spark.

Since the same assumptions that the high-order functions themselves are side effect free also hold for filter and map implemented in Java, they were already declared as pure in the native declarations. This leads to the property that if they are passed a pure function, the whole construct won't have any side effects. If they are passed an impure function, the compiler will throw an error, making it impossible to execute an impure function with filter, map or reduce.

The next hypothesis would be that it should be possible to pass a function from Frege to the Java part and let Apache Spark run it.

The corresponding native declarations to pass a function to the Java part and receive back an Apache Spark Function would look as follows:

```
pure native create   "bindings.FunctionHelper.createFunction" {a,b}       :: (a -> b) ->
Function a b
```

bindings.Function.fr

To further understand the task at hand, it is necessary to have a look at what a function is on both sides, Frege and Apache Spark.

Frege Functions

In Frege, a simple function has a type declaration such as (a -> b), or in a more explicit example (String -> Double). It takes a value of type a and returns a value of type b, or takes a String and returns a Double respectively. As already introduced in the chapter on Frege and by investigating the Java class to which the Frege-code is compiled, it is clear that a Frege-function like (a->b) eventually is compiled to an Java-Object of type Func.U.

```
…
public class Func  {
  @FunctionalInterface public interface U<𝒜, ℬ>
    extends Lazy<Func.U<𝒜, ℬ>>, Kind.U<Func.U<𝒜, ?>, ℬ>, Kind.B<Func.U<?, ?>, 𝒜, ℬ>
   {
    public Lazy<ℬ> apply(final Lazy<𝒜> a) ;
    public default Func.U<𝒜, ℬ> call() {
      return this;
    }
    public default boolean isShared() {
      return true;
    }
  }
…
}
```

frege.run8.Func.java

Functions in Apache Spark

On the other side, in the Apache-Spark part written in Java, the function passed as parameter to the filter and map functions are of type org.apache.spark.api.java.function.Function.

```
// licence information omitted

package org.apache.spark.api.java.function;

import java.io.Serializable;

/**
 * doc comment omitted
 */
@FunctionalInterface
public interface Function<T1, R> extends Serializable {
  R call(T1 v1) throws Exception;
}
```

org.apache.spark.api.java.function.Function.java

Those two types are as expected conceptually similar as they both intend to represent a function. They even both are functional interfaces and have a method named call(...), but they are different in their implementations.
Since it is not possible the retrieve the content of a function back out of a Func.U-object, the goal is to create a org.apache.spark.api.java.function.Function that executes the frege.run8.Func.U in its call-method.

```
public static <A, B> Function<A, B> createFunction(){
  return new Function<A, B>() {
    public B call(A value) {
      return (B) value; // TODO
    }
  };
};
```

Integration between Frege and Apache Spark Function Types

A simple Frege function that appends a “y” to an existing String looks as follows:

```
append :: String -> String
append x = x ++ "y"
```

snippets.Tryout.fr

Such a function can be passed as an argument via a corresponding native declaration

```
data Function a b = pure native "org.apache.spark.api.java.function.Function" where

  pure native create   "bindings.FunctionHelper.createFunction" {a,b}       :: (a -> b) ->
Function a b
```

bindings.Function.fr

whereas the code to call that function is written as follows:

```
mappedData = currentData.map (Function.createFunction append)
```

examples.IntegrationRDDExample.fr

The append function will then be transformed into a Func.U and can be executed in Java as follows, given an input String x and a Func.U f:

```
String result = f.apply(x).call();
```

As the apply-method requires a parameter of type Lazy<A>, it is necessary to wrap the A value x in a Lazy wrapper. This can be done by calling Thunk.lazy(x).

The combination of the two code snippets above that would lead to the intended behavior is executing the Func.U on the parameter passed to the Function’s call method.

```
public static <A, B> Function<A, B> createFunction(Func.U<A, B> f) {
  return new Function<A, B>() {
    public B call(A x) {
      return (B) f.apply(Thunk.lazy(x)).call();
    }
  };
};
```

bindings.FunctionHelper.java

Functions with multiple arguments

In Frege, a function that takes two values of type Double and returns a new Double, such as summing the two values up, has a type signature of Double -> Double -> Double.
As explained in chapter "Functions in Apache Spark", functions with multiple arguments will be mapped to the corresponding Func types.
In the case of two parameters, functions get compiled to a construct of type Func.B<A, B, C>, which is executed slightly different.

```
return (C) f.apply(Thunk.lazy(x), Thunk.lazy(y)).call();
```

For this kind of functions with multiple arguments, Apache Spark also has different wrappers such as Function2 which work analogously.

```
@FunctionalInterface
public interface Function2<T1, T2, R> extends Serializable {
  R call(T1 v1, T2 v2) throws Exception;
}
```

org.apache.spark.api.java.function.Function2.class

Therefore, the adjusted code for Functions with two parameters can be implemented as follows:

```
public static <A, B, C> Function2<A, B, C> createFunction2(Func.B<A, B, C> f) {
        return new Function2<A, B, C>() {
                public C call(A x, B y) {
                        return (C) f.apply(Thunk.lazy(x), Thunk.lazy(y)).call();
                }
        };
};
```

bindings.Function2Helper.java

Further versions of the different multi-parameter functions, analogously to the different versions of the Func.U and org.apache.spark.api.java.function.Function3<T1, T2, T3, R> would have to be implemented to make more complex examples work.

Possible Advantages of this Approach

This would close the integration between Frege and Apache Spark in a very straightforward way and allow to pass Frege functions to Apache Spark to be executed.

Benefits of this approach mainly come from working with pure functions. The Frege compiler will enforce that function that are effectful are declared as such and can only be used in corresponding contexts, which avoids hidden errors.

```
pureFunction :: Double -> Double
pureFunction x = x * 10.0

impureFunction :: Double -> IO Double
impureFunction x = do
  putStrLn "This function prints something and is impure!"
  return (x * 10.0)
```

functions.FunctionPool.fr

Pure functions themselves tend to be concise and easy to read which improves clarity when working with them as boilerplate code is reduced.
Furthermore, the type signatures of functions are meaningful, and it is possible to reason about the program just by looking at the type signatures. Hidden state changes or IO are made visible to the developer as type signatures also indicate possible side effects.
Having the guarantee of no side-effects can then be beneficial when running calculations in parallel or concurrently on multiple nodes. Through the use of pure functions, it is guaranteed that the computations will compute the same result regardless of which node they are executed on.
In any modern application, there inevitably will be impure code, for example for reading an input file, but this code is pushed to the edge of the application. Using a functional programming approach keeps the core functionality of the application pure which has the potential to make the core functionality more stable and predictable.
Since the functions themselves are clearer, testing is also easier. Furthermore, functional programming testing approaches like property-based testing can be applied to test the functions. This allows to test the scope of all possible values including possible corner cases and could also result in more stable and bugless code.

Issue with Serialization and Serializable Functions

As can be seen so far, from a conceptual standpoint, it is possible to map the two different types of function used in Frege and Apache Spark together. The code compiles without errors, but Apache Spark runs into an exception when executing it.

```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
…
Caused by: java.io.NotSerializableException:
examples.frege.numbers.IntegrationExampleFrege$$Lambda$38/1209962934
Serialization stack:
- object not serializable (class:
examples.frege.numbers.IntegrationExampleFrege$$Lambda$38/1209962934, value:
examples.frege.numbers.IntegrationExampleFrege$$Lambda$38/1209962934@6cd56321)
- field (class: bindings.Functions$3, name: val$fStored, type: interface frege.run8.Func$U)
- object (class bindings.Functions$3, bindings.Functions$3@6f139fc9)
- field (class: org.apache.spark.api.java.JavaPairRDD$$anonfun$toScalaFunction$1, name:
fun$1, type: interface org.apache.spark.api.java.function.Function)
- object (class org.apache.spark.api.java.JavaPairRDD$$anonfun$toScalaFunction$1,
<function1>)
…
```

For the full stack trace, please refer to the appendix.

Remark

The stack trace includes the serialization stack from Apache Spark's SerializationDebugger. This tool searches the object graph in the case of a NotSerializableException to help find the object that cannot be serialized. [18]

It can be derived from the exception that it is the Func.U object that can't be serialized. Apache Spark expects the passed object, which in this case is the Java object equivalent of a Frege Function, to be serializable, that is why org.apache.spark.api.java.function.Function extends Serializable. But the Func.U-object used inside the Apache Spark Function is not serializable.

Serializable

> Via Java Serialization you can stream your Java object to a sequence of byte and restore these objects from this stream of bytes. [19]

Serialization helps to persist objects or send them over networks. In our case, Apache Spark will send the tasks including the function to be run to the executors to distribute the work, therefore it is required that the function is serializable.
In Java, the marker interface java.io.Serializable can be extended to mark an object as serializable.

> A marker interface is an interface that **has no methods or constants inside it**. It provides **run-time type information about objects**, so the compiler and JVM have **additional information about the object**. [20]

Serializable Functions and Apache Spark

When an Apache Spark application is run in cluster mode, everything needed to execute the calculations is packed up, serialized and sent to the executors on the worker nodes to be run. This is why Apache Spark enforces the Function to be serializable and will throw a NotSerializableException if this is not the case. [21]

The Function-classes (org.apache.spark.api.java.function) used by Apache Spark are defined to extends java.io.Serializable and are serializable by default as expected.
The stack trace indicates that the underlying field of type frege.run8.Func$U is not serializable. As demonstrated earlier, Func.U corresponds to the way functions are represented in Frege to interoperate with the JVM.

The interfaces in the class Func don't extend Serializable by default, as the Serialization constraint was not a requirement in the Frege context. Therefore, the resulting objects are not serializable which leads to the NotSerializableException when a Func-object is called inside an Apache Spark Function.

Attempts to Solve the Serialization Problem

The following options were pursued to solve or circumvent the serialization problem.

Implement Serializable on interface (in the frege runtime/language)

Since it is the Func.U-Object that is not serializable, the obvious approach would be to define the object as serializable. In Java, this is done by implementing the Serializable marker interface. Defining the high-level Func.U- interface to implement Serializable would mean to make changes at the core of the Frege language. Func.U is part of the frege.run8-package, which belongs to the Frege runtime. Changing the language to adjust to the need of Apache Spark seemed a radical approach and was not further pursued, especially given the fact that serializing functions is by itself a controversial thing to do. Furthermore, a quick experiment to try to change the Frege runtime locally was aborted, as it was not possible to find a way to solve the occurring errors when trying to build the compiler from the latest sources.

Make serializable wrapper interface

Another option was to build a wrapper interface that implements Serializable around the interface Func.U. This probably would have had the same effect as having the Func.U interface implement Serializable, but this did not work, as the nested objects of type Lazy and Kind are not serializable as well. Further refactoring those classes would have meant to rebuild and circumvent the whole Frege runtime which seemed not reasonable. Therefore, since the passed function type and its underlying elements are not serializable, it is not easily possible to create a serializable wrapper.

Define fields as transient

Serialization is often used in the context of storing data in a database. If an object can't be serialized because one or multiple of its fields are not serializable, it is possible to mark the affected fields as transient to prevent them from being included in the serialization of the parent object.
This won't work in this context, as all the fields in Func.U are relevant and cannot be left out by defining them transient when serializing the Func.U object, as else a NullPointerException is thrown.

Cast & Serialize (SerializableLambda-trick)

There exists a trick to make a lambda in Java serializable by casting an object and simultaneously adding Serializable to the cast. This trick works for simple cases, but unfortunately doesn't yield the desired result as the casting returns a "cannot be cast to java.io.Serializable"-Exception. The reason is the same as for the serializable wrapper interface in that the underlying Lazy and Kind types of Frege's Func.U as nested fields are not serializable. [22]

Extend externalizable & Using custom serialization

Another possibility is to overwrite the default serialization behavior to make sure the objects are serializable. This can be achieved by extending java.io.Externalizable. The object would then have to implement the additional methods readExternal(ObjectInput in) and writeExternal(ObjectOutput out) to define a custom serialization. [23]
This approach was not further pursued due to time and topic constraints.

Alternative Approaches

As it was not possible to find a solution to properly solve the serialization issue, other approaches were tested to be able to further integrate Apache Spark and Frege and work around the serialization issue. The main idea evolves around not passing a function as an argument to the nodes but to find a different mean of passing the functionality or an identifier thereof to the node, even if this means to make the function available at the nodes as well.

Interpreted Functions

Since the Frege function is not a serializable object once it is compiled by the Frege compiler, the idea was to pass the function in another serializable form. The most straightforward option would be to pass Frege source code directly instead of the compiled function as a parameter and then interpret and execute the Frege code on the nodes. Different to the Func-objects, the source code of a Frege function is a String and therefore serializable by default, as the class String implements Serializable.

For this, the following incremental milestones were set.

- Java example on how to use the Frege interpreter to execute a script
- Find a way to pass the Frege script as a parameter, load it to the interpreter and execute it.
- Integrate Frege scripting into Apache Spark by creating an Apache Spark Function that receives Frege script and returns the executed result, also on a cluster node.
- Instead of passing a function in script form, an alternative option would be to pass the function name as a parameter and use the scripting engine to load functions defined in a Function Pool and calling them by name.

To understand the task at hand, have a look at the involved technologies and concepts.

Concepts

The goal is to directly execute Frege source code instead of running the compiled bytecode generated from Frege source code. The normal use case for Frege would be to have a compiler that compiles the Frege source code to Java which then gets compiled to Java Bytecode which can be executed on the JVM.
However, there is another tool that can process Frege source code as there exists an interpreter for Frege.
A compiler creates files containing machine code which can be executed while an interpreter runs through source code line-by-line, analyses and executes it at runtime. [24]

Interpreter

The Frege interpreter itself is written in Frege, interprets the given scripts and adjusts the internal state accordingly. The Frege Interpreter can be found on Github under the following URL: https://github.com/Frege/frege-interpreter

Tools that intent to run Frege at runtime like scripting make use of the interpreter. For example the Frege REPL (Read-Eval-Print-Loop) and the Online REPL use the interpreter to evaluate and execute Frege scripts. [25]

JSR223 Scripting for the Java Platform

JSR, which is short for Java Specification Request, is a process to get new requirements or other changes for the Java language. [26] There exists a JSR that dealt with the ability to support scripting in the JVM. The easiest way to use the scripting ability from Java is with the help of the JSR223.
JSR223 is a scripting API for JVM languages. Through this framework, Java applications can host scripting engines. [27]

Scripting in Frege

There exists an implementation of a script engine in Frege to fulfill the JSR223, which underneath is running the Frege Interpreter to execute scripts.
The corresponding files can be found here:
https://github.com/Frege/frege-interpreter/blob/master/frege-interpreter-core/src/main/frege/frege/scriptengine/FregeScriptEngine.fr

Usage of Frege interpreter – Java Example

Since Frege supports the JSR223, it is possible to execute scripted Frege code with the help of a scripting engine. The following Java code illustrates this. [28]

```java
final ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine frege = factory.getEngineByName("frege");
// scripting engine evalutes '1 + 1', result is 2
int     a = (Integer) frege.eval("1+1");
assert (a == 2);
System.out.println(a);

// load the function f into the context
frege.eval("f x y = x + y");
// execute function on parameters
int b = (Integer) frege.eval("f 1 2");
assert (b == 2);
System.out.println(b);
```

examples.InterpreterExample.java

Scripting Frege in Apache Spark

Apache Spark has no active support for JSR223, therefore the functionality to use Frege's ScriptingEngine has to be added another way. Given the acquired knowledge so far, the simplest approach seemed to write a generic Apache Spark function that internally executes the Frege ScriptEngine to return a result.

Passing Frege Scripts as parameter

This version includes a Frege script that is passed as a parameter to Apache Spark. The native declaration and the code to call it looks as follows:

```
mappedData <- parsedData.map $ Function.createInterpretScriptFunction "f x = x * 10.0"
```

examples.InterpretationRDDExample.fr

```
pure native createInterpretScriptFunction
"bindings.FunctionHelper.createInterpretScriptFunction" {a,b} :: (String) -> Function a b
```

bindings.Function.fr

Java helper function to create an Apache Spark function:

```
public static <A, B> Function<A, B> createInterpretScriptFunction(String
functionScript) {
     return new Function<A, B>() {
          public B call(A x) throws IOException, ScriptException {
               return (B)
(ScriptExecutor.loadAndExecuteScriptFunction(functionScript, x));
          }
     };
};
```

bindings.FunctionHelper.java

The code to interpret and execute the Frege Script on the worker node:

```
public static void loadFunction(String function) throws IOException,
ScriptException {
        loadScriptEngine();
        frege.eval(function);
}

public static <A, B> B executeScriptFunction(A x) throws IOException,
ScriptException {
        return (B) frege.eval("f " + x.toString());
}

public static <A, B> B loadAndExecuteScriptFunction(String function, A x)
throws IOException, ScriptException {
        loadFunction(function);
        return executeScriptFunction(x);
}
```

script.ScriptExecutor.java

First, the scripting engine gets checked and properly loaded if it is not set up yet, then the script function passed as a parameter gets evaluated and loaded into the interpreter context. Afterwards, the function gets called on the data passed to the function.

Analysis

This approach yields the desired result in that it executes a Frege function on the worker nodes, but there are drawbacks to this method:

- The Frege script inside the String is not checked by the compiler at compile time, therefore type safety is not guaranteed which may result in runtime errors. Furthermore, loading a script as a one-liner has certain limitations. This gives away some of the possible benefits the Frege compiler would be capable of.
- Dedicated handling of the function names has to be provided, as the name of the functions has to be known to execute it. In this simple case, the function has to be defined under the name “f”, as the function “f” is called on the data. This is not feasible in more complex examples and another mechanism would have to be implemented.
- Unnecessary parsing is applied as the data types provided to the Function are parsed to a String to pass it to the interpreter, and the result is then parsed back to the result type.

Providing Frege code from the source

To tackle some of these drawbacks encountered in the previous approach, it would also be possible to collect all functions in a single file and just pass the function names as an identifier instead of passing single functions as parameter. This for example would allow the source code file to be checked at compile time.
But in the JVM, only the bytecode is available at runtime and executed. To execute Frege scripts at runtime, the source code of these scripts would be needed at runtime. And since it is not possible to directly access the source code of a function in the JVM, another mechanism has to be used to get an executable Frege-script. In Java, the generated class-files are merged into a JAR (Java Archive), which is a zipped file with the ending *.jar. One possible way is to not just put the bytecode into the archive file, but to keep the source code and the compiled class-files together. By adding the source files, namely Frege's *.fr files, the source code of the Frege functions is also included in the jar and therefore available at runtime.

Decompilation would also have been a possible option to get the source code, but there exists no Frege Decompiler and adding the original sources to the JAR seemed a much more reasonable approach.

The following code will browse the content of the JAR at runtime and load the Frege module FunctionPool.fr containing the Frege functions into the interpreter context.

```
public static void loadFunctions() throws IOException, ScriptException {
	if (frege == null) {
		loadScriptEngine();
		JarFile jarFile = new JarFile("frege-spark.jar");

		final Enumeration<JarEntry> entries = jarFile.entries();
		while (entries.hasMoreElements()) {
			final JarEntry entry = entries.nextElement();
			if (entry.getName().contains(".")) {
				if (entry.getName().contains("FunctionPool.fr")) {
					JarEntry fileEntry = jarFile.getJarEntry(entry.getName());
					InputStream input = jarFile.getInputStream(fileEntry);
					InputStreamReader isr = new InputStreamReader(input);
					BufferedReader reader = new BufferedReader(isr);
					frege.eval(reader);
					reader.close();
				}
			}
		}
		// import entry point
		frege.eval("import functions.FunctionPool");
		jarFile.close();
	}
}
```

script.ScriptExecutor.java

In an intermediate step, every function was searched and loaded on the fly, but loading the complete FunctionPool.fr initially from the BufferedReader and then just execute the scripts seemed much more reasonable. This is why in the final version, the complete FunctionPool is loaded into the Frege scriptEngine.

The resulting application that uses interpreted functions could look as follows:

```
module examples.InterpretationRDDExample where

--imports omitted

main :: IO ()
main = do
  sparkConfig <- SparkConf.new ()
  list <- arrayFromListST [applicationJar, fregecJar, interpreterJar] --
  sparkConfig.setAppName "Frege-Spark"
  sparkConfig.setJars list
  sparkConfig.setMaster "local"
  --sparkConfig.setMaster distributedSparkMaster
  sc :: MutableIO JavaSparkContext <- JavaSparkContext.new sparkConfig

  currentData <- JavaSparkContext.textFile sc "data/first.csv"
  parsedData = currentData.map CustomFunction.Function.convertToDouble
  filteredData = parsedData.filter $ Function.Function.createInterpretedFunction
"filterThreeOrFive"
  mappedData = parsedData.map $ Function.Function.createInterpretedFunction "timesTen"
  sum = mappedData.reduce Function2.createInterpretedFunction2 "sum"
```

examples.InterpretationRDDExample.fr

Loading modules and handling imports

Currently, the ScriptExecutor loads all files with a name ending with FunctionPool.fr and imports functions.FunctionPool as an entry point.

Imports in context of the interpreter proved somewhat troublesome, as the imports in the FunctionPool module seem not to work, but imports directly into the Frege interpreter, by calling “import functions.NestedImportFunctionPool” for example, work as expected. For keep the integration example as simple as possible, the import cases won’t be further addressed.

Problems with imports may lead to errors such as:

```
19/09/03 17:56:09 ERROR Executor: Exception in task 0.0 in stage 1.0 (TID 1)
frege.runtime.Undefined: [
1: E <console>.fr:1: Error:/functions/FunctionPool.java[25:17]: cannot find symbol
  symbol:   class NestedImportFunctionPool
  location: package functions
]
at frege.prelude.PreludeBase.error(PreludeBase.java:16141)
at frege.scriptengine.FregeScriptEngine.evalResult(FregeScriptEngine.java:1023)
at frege.scriptengine.FregeScriptEngine.lambda$eval$53(FregeScriptEngine.java:1363)
at frege.prelude.PreludeBase$TST.lambda$performUnsafe$4(PreludeBase.java:10446)
at frege.run8.Thunk.call(Thunk.java:231)
at frege.scriptengine.FregeScriptEngine$JFregeScriptEngine.eval(FregeScriptEngine.java:637)
at frege.scriptengine.FregeScriptEngine$JFregeScriptEngine.eval(FregeScriptEngine.java:650)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:249)
at script.ScriptExecutor.loadFunctions(ScriptExecutor.java:110)
at script.ScriptExecutor.executeLazyFunction(ScriptExecutor.java:30)
at bindings.FunctionHelper$2.call(FunctionHelper.java:29)
```

Analysis

So far, it is possible to load Frege functions into the interpreter and call the previously defined functions by their name.

The resulting Function is serializable, can be passed to the nodes in an Apache Cluster and will not throw a NotSerializableException. This now allows to run an Apache Spark application from Frege and use a Frege functions to filter, map or reduce through the data using the RDD-specific functions through native declarations, also on a distributed node.

This now adds additional benefits compared to interpreting single scripts. The Frege code in the FunctionPool is checked at compile time which adds additional safety as any developer will be warned by the compiler of possible type errors.
Inside Frege it is possible to work in a pure context, write and test pure functions that build that core of the application.

There still are certain general disadvantages to this approach:

- Using an interpreter still leads to unnecessary parsing of the data. The interpreter takes a String, therefore the data will be parsed from their generic type A to a String an the result is passed back as an object which has to be parsed to the generic result type B.
- This approach introduces the Frege Interpreter as a new part to the system, which adds complexity and further possible roots for errors and restrictions such as for imports.
- The name of the function is passed as a String instead of a function reference, which exits the Frege type system and makes for another source of errors that will only be revealed at runtime. For example, a simple mistyped name would lead to a runtime error. This clearly violates the intention to have the compiler avoid runtime errors. There is also a certain detachment of the source code that was checked and compiled and the String from the source files that is used as a script, even though it still is the same code. Further information on this topic and on giving away type safety can be found in the next chapter “Possible Runtime Exceptions instead of Compile Checks”.
- In the current version, loading the script into the scripting engine happens multiple times. It should be further investigated whether this is redundant and how it could be improved.
  Preloading the script once outside of the Function<A, B>(String function) won’t work, as the function then won’t be available in the scripting engine of the worker node. Therefore, it must be invoked inside the function body and cannot be preloaded beforehand.
- First impressions conclude that using the interpreter has performance drawbacks, at least for the use case of the minimal amount of data where initiation time is crucial. Further thoughts about performance can be found in Chapter “Further Remarks – Note on Performance”
  It is important to note that these performance implications are most likely due to the use of the Frege interpreter, not the Frege language itself. If another solution helps to solve the serialization issue and make Frege Functions executable on Apache Spark, these implications probably wouldn’t apply.

Nevertheless, this approach proves that integration between Frege and Apache Spark is possible to a level where functions written in Frege can be executed on a worker node in a distributed Apache Spark application.

But the mechanism to run the Frege functions clearly differs when running a function on the interpreter compared to passing a frege.run8.Func.U-object as initially intended.
Therefore, the conclusion that it is possible to run a Func.U-object on a distributed Apache Spark worker node is not yet given and would have to be further investigated.

Possible Runtime Exceptions instead of Compile Checks

Through the interpreter it is possible to run Frege functions on distributed Apache Spark nodes, yet it bypasses some of the intended constraints and benefits of using functional programming in the first place.
In an ideal case the compiler could detect type issues already on the source code. Running the code on the interpreter won't make direct use of this compiler checks, although they are done on the FunctionPool. But the calling of functions on the interpreter may lead to runtime exceptions.
When calling a Frege function by name and the given name is wrong or the function does not exist, a NullPointerException will be thrown. This is a runtime exception and something that should be prevented, but the approach to use the Frege interpreter allows for this error to happen.

```
java.lang.NullPointerException
at frege.compiler.grammar.Lexer.lambda$lex$27(Lexer.java:2945)
at frege.run8.Thunk.call(Thunk.java:231)
at frege.compiler.grammar.Lexer.lex(Lexer.java:2946)
at frege.compiler.grammar.Lexer.lexer(Lexer.java:3349)
at frege.compiler.grammar.Lexer.passCS(Lexer.java:3447)
at frege.interpreter.FregeInterpreter.lambda$null$117(FregeInterpreter.java:6242)
at frege.run8.Thunk.call(Thunk.java:231)
at frege.control.monad.State.lambda$promote$0(State.java:1827)
at frege.run8.Thunk.call(Thunk.java:231)
at frege.prelude.PreludeBase$TST.lambda$$gt$gt$eq$3(PreludeBase.java:10440)
…
at frege.run8.Thunk.call(Thunk.java:231)
at frege.scriptengine.FregeScriptEngine.lambda$eval$53(FregeScriptEngine.java:1359)
at frege.prelude.PreludeBase$TST.lambda$performUnsafe$4(PreludeBase.java:10446)
at frege.run8.Thunk.call(Thunk.java:231)
at frege.scriptengine.FregeScriptEngine$JFregeScriptEngine.eval(FregeScriptEngine.java:637)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:264)
at script.ScriptReader.executeFunctionFromJar(ScriptReader.java:86)
at bindings.FunctionHelper$2.call(FunctionHelper.java:36)
…
```

Furthermore, the compiler would warn if an impure function was called in a pure context. Unfortunately, by calling a function through the interpreter, this direct linkage is not given anymore. The interpreter will only throw a runtime exception in such a case.

```
frege.runtime.Undefined: [37: E <console>.fr:37: IO Double is not an instance of
    Real]
at frege.prelude.PreludeBase.error(PreludeBase.java:16141)
at frege.scriptengine.FregeScriptEngine.evalResult(FregeScriptEngine.java:1023)at
frege.scriptengine.FregeScriptEngine.lambda$eval$53(FregeScriptEngine.java:1363)
at frege.prelude.PreludeBase$TST.lambda$performUnsafe$4(PreludeBase.java:10446)
at frege.run8.Thunk.call(Thunk.java:231)
at frege.scriptengine.FregeScriptEngine$JFregeScriptEngine.eval(FregeScriptEngine.java:637)
at frege.scriptengine.FregeScriptEngine$JFregeScriptEngine.eval(FregeScriptEngine.java:650)
at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:249)
at script.ScriptExecutor.loadFunctions(ScriptExecutor.java:110)
at script.ScriptExecutor.executeLazyFunction(ScriptExecutor.java:30)
at bindings.FunctionHelper$2.call(FunctionHelper.java:29)
…
```

Therefore, while the compiler still checks the FunctionPool, not all checks it is capable of can be fully enforced as the interpreter is added as an intermediate layer.

In an ideal case, if an impure function with an according native declaration is passed to map or filter, a compiler error should be thrown.

```
native createIOFunction   "bindings.FunctionExperimentsHelper.createIOFunction"  {a,b} :: ()
-> IO (Function a b)
```

bindings.custom.CustomFunction.fr

```
mappedData = parsedData.map $ Function.createIOFunction ()
```

examples.IOIntegrationExample.fr

```
-type error in expression createIOFunction () type is : IO (Function t2 t1) expected:
Function t1 t2
```

As shown here, the compiler would have the capabilities to detect impure functions passed to an otherwise pure context, which cannot be fully used in the current state of the integration when using an interpreter.

```
pureFunction :: Double -> Double
pureFunction x = x * 10.0

impureFunction :: Double -> IO Double
impureFunction x = do
  putStrLn "This function prints something and is impure!"
  return (x * 10.0)
```

functions.FunctionPool.fr

By using the interpreter, this mechanism can be tricked by calling an impure function anyway which will result in a runtime error. But by implementing the integration between Frege and Spark straightforward, this mechanism would work as expected, and the compiler would give warnings if an impure function was passed as calculation.

Other Alternative Approaches

Using Java Reflection on the compiled Frege module

Another option would be to use Reflections as the executing mechanism. The idea behind is that the function code would already be available on the worker node, just the mechanism to call it would be different compared to the interpreter approach.
There are again different variations on how to do it:

- Create a Java Function Pool and call the functions via Reflections from Frege
- Call the Java Functions compiled from the Frege Function Pool and execute them through reflection

```
filterThreeOrFive :: Double -> Bool
filterThreeOrFive 3.0 = true
filterThreeOrFive 5.0 = true
filterThreeOrFive _ = false
```

functions.FunctionPool.fr

```
final public static boolean filterThreeOrFive(final double arg$1) {
  if (3.0D == arg$1) {
    return true;
  }
  if (5.0D == arg$1) {
    return true;
  }
  return false;
}
```

functions.FunctionPool.java (compiled output of functions.FunctionPool.fr)

Since the compiled code works with primitive types, it has to be ensured that the generic method is called on the primitive types as well. For example `getMethod(functionName, Double.class)` wouldn't work and throw an MethodNotFoundException.

```
public static <A> Function<A, Boolean>
createReflectionFilterFunction(String functionName) {

        return new Function<A, Boolean>() {
                public Boolean call(A x) throws NoSuchMethodException,
SecurityException, IllegalAccessException, IllegalArgumentException,
InvocationTargetException, ClassNotFoundException {
                        Method method =
Class.forName("functions.FunctionPool").getMethod(functionName,
double.class);
                        Double value = (Double) x;
                        Boolean result = (Boolean) method.invoke(null,
value);
                        return result;
                }
        };
};
```

bindings.FunctionExperimentHelper.java

```
data Function a b = pure native "org.apache.spark.api.java.function.Function" where

  pure native createReflectionFilterFunction
"bindings.FunctionExperimentsHelper.createReflectionFilterFunction"              {a} ::
(String) -> Function a Bool
```

bindings.custom.CustomFunction.fr

```
filteredData = parsedData.filter $ Function.createReflectionFilterFunction
"filterThreeOrFive"
```

examples.ReflectionRDDExample.fr

Technically, using reflections also works, even in a distributed mode as long as the class files of the FunctionPool are available in the application JAR which is run on Apache Spark. But this approach still fights with the issues of having possible runtime errors and the troublesome parsing between primitive and wrapper classes in this simple case as well as with the generic types in the general case.

Therefore, this approach was not further pursued to stay more closely in the context of Frege.

Using Java Native Module

Another option would be to use a Java Native Module in Frege and write Java code in a Frege module. In the simplest case this would look as follows:

```
pure native createNativeFunction Function.NativeFunction.createNativeFunction {a,b} :: () ->
Function a b

native module where {
  public static class NativeFunction {
        public static <A, B> org.apache.spark.api.java.function.Function<A, B>
createNativeFunction() {
                return new org.apache.spark.api.java.function.Function<A, B>() {
                        public B call(A x) {
                                return (B) x;
                        }
                };
        };
  }
}
```

bindings.Function.fr

This approach quickly yields issues, as there are constraints with regard to generics enforced by the compiler.

```
Multiple messages at this line.
-An anonymous class cannot subclass the final class Function
-The method createNativeJavaFunction() is undefined for the type
 NativeModuleRDDExample.NativeFunction
-Java compiler errors are almost always caused by bad native declarations. When you're sure this is
 out of the question you've found a compiler bug, please report under https://github.com/frege/frege/
 issues and attach a copy of /Users/dkm/frege-workspace/frege_spark/target/classes/examples/frege/
 numbers/TryoutIntegrationExampleFrege.java
-The type Function is not generic; it cannot be parameterized with arguments <Double, Boolean>
```

Running the application then on a remote worker yields the following exception:

```
java.lang.ClassNotFoundException:
examples.frege.numbers.TryoutIntegrationExampleFrege$NativeFunction$2
at
org.apache.spark.serializer.JavaDeserializationStream$$anon$1.resolveClass(JavaSerializer.sca
la:67)
```

But this could be resolved by making sure that corresponding classes all were added properly to the frege-spark.jar. Please refer to Appendix B – Instabilities and Troubleshooting - Issues with frege-spark.jar to read about how to resolve these issues.

While this approach generally also works in a distributed context, it doesn't solve the problem of wanting to execute an arbitrary Frege function. But it is possible to write explicit functions this way, as long as you manage to work around the issues that arise in context of Generics and the parsing between the types.

Adjusting the Frege language and compiler

Another possible but very different way would be to model the serialization constraint into the context of Frege by making adjustments to the Frege language or the compiler. As already proposed in an attempt to solve the serialization issue, functions could be implemented in a different way. One option would be to make sure that the existing frege.run8.Func.U-class and its subclasses are serializable, and another option would be to model the Serialization constraint into the language.
It might be possible to create a Monad data-type like Ser(…) which would than introduce and map the additional constraint into Frege such as Ser(a-> b) -> [a] -> [b]

This is only a sketch of a general idea and such changes would run through the whole language and application. An implementation thereof would very likely affect readability and clarity of code in a negative form, and as other options presented easier-to-use integration solutions, this approach was not further pursued.

More information about such an approach could be found in the following projects where similar ideas were pursued:

- Software Transactional Memory,
  http://www.frege-lang.org/doc/frege/control/concurrent/STM.html
- FregeFX
  https://github.com/Frege/FregeFX

Conclusion

This thesis worked out the theory how Frege interoperates with Java and how this knowledge can be used to write Apache Spark applications in Frege. With an iterative development approach from known ground towards the unknown, the first step was to create a simple Apache Spark application in Java to ensure a working baseline example. Through progressive integration, different Apache Spark applications written in Frege were developed, which showcase various ways and levels of how these two technologies can interoperate with each other.

The first iteration step included an application written completely in Frege that calls Java methods through native declarations for all of the functionality underneath. Through bindings between Frege and Java, existing Apache Spark APIs were made available in Frege and any custom function or object needed was deferred to dedicated Java code and called in Frege through further native declarations.

A proof-of-concept showed that data from an RDD can also be fetched from Apache Spark and worked with purely in Frege, with transformation functions written in Frege. This lead to the next step of working on an example where Frege functions get executed on Apache Spark. This constitutes the highest level of integration between Frege and Apache Spark and allowed to investigate benefits of using a functional programming language like Frege in the context of Apache Spark.

This thesis showed that Frege and Apache Spark theoretically could intertwine directly from a conceptual point of view, as the theory has been worked out on how both these technologies map the concept of functions in their respective context. But the straightforward solution, while theoretically feasible, was not realizable in practice. Apache Spark enforced further constraints at runtime which required the passed function objects to be serializable. This was not the case for the object that represents a function in Frege and therefore lead to serialization errors at runtime. As trying to solve these issues with different approaches did not yield a satisfying solution, further ideas were investigated as alternatives to provide the integration between Frege and Apache Spark on a level where Frege functions could be executed on a distributed Apache Spark node.

The idea to use the Frege interpreter was chosen as it presented a way to pass functions in a serializable form. This approach delivered a working example to prove the feasibility of the idea and enabled having pure Frege functions executed on distributed Apache Spark nodes. But a different mechanism in comparison to the intended straightforward approach is used and this solution has some drawbacks. Using an interpreter opens the possibility to circumvent some of the safety measures that using a functional programming language like Frege could provide.

Finally, the theoretical elaboration of the results of this thesis also demonstrated how integrating Frege and Apache Spark straightforward would yield the desired benefits of working with pure functions and having the compiler guarantee type safety and the absence of side effects to reduce errors in the code.

Further Remarks

Note on instabilities

Instabilities with mismatching versions, issues with the different components like the scripting engine and errors as well as performance issues with the eclipse IDE strongly hindered development. These kinds of issues are to be expected when working with a niche programming language, as the development of the associate tools is limited. Please read the section “Instabilities” in the appendix to learn more about issues encountered when working with Frege and Apache Spark.

Note on performance

In the scope of this thesis, the performance implications were not further investigated, but as performance is crucial when working in a Big Data context, the impact on evaluation speed and performance should definitely be further evaluated.
From a first intuition, it seems that using an interpreter and loading function slowed down the example applications at hand, but this was not further explored.
For evaluating performance of a big data system, it should be kept in mind that loading time may have a bigger implication on a small data set while longer execution time may have a bigger influence when running on large amounts of data.

Note on the different Apache Spark API

To support the different API’s like RDD and DataSet, it was necessary to write native declarations for the needed methods of both those classes and all the auxiliary methods. Additional to the classes needed by the examples working with a RDD, native declarations also had to be implemented for the following classes to ensure integration with the DataFrame-API:

DataFrameReader, DataFrameWriter, DataSet, Row, SQLContext, FilterFunction, MapFunction, ReduceFunction, Encoder

Note on building the Frege compiler

As one of the possible solutions to fix the TaskSerialization-Problem would have been the adjust the Frege language to make Func.U serializable, a limited sidetrack included trying to build the Frege compiler. For unknown reasons, doing so resulted in a “code too large”-Error when building the compiler in the latest version, 3.25.84. This may be due to a particularity of the development computer used and was not further investigated due to the constraints of this project. This development track was not further pursued, as changing the Frege language to solve the problem at hand was not considered an appropriately feasible approach. Nevertheless, it would have been interesting to see whether adjusting the language would have yielded the desired results.

Future work

Implement further parts of the Apache Spark API

Native Declarations have only been provided for small parts of the Apache Spark APIs to prove the basic interoperability between Frege and Apache Spark. To use more of the methods and features of Apache Spark, further parts of the API would have to be worked out. Adopting further methods and classes could be done as shown on the basic functionality in this thesis.

Find a better solution to solve the serialization problem

Although an interesting approach to pursue and learn more about the ecosystem at hand, the solution found in this thesis is not completely satisfactory. Using an interpreter was the result of trying to find an alternative way to make a function serializable. Theoretically, this thesis showed that Frege's function elements and Apache Spark Functions could be mapped straightforward. Therefore, the possibility remains open that it might be possible to solve the serialization problem in some other way, but with the given knowledge and the scope of this project, this was not achieved. If there is a way to make the passed Frege function serializable, this would be preferential as a more straightforward solution. Furthermore, solving the integration straightforward as proposed in this thesis would in contrast to the integration with an interpreter make full use of the potential of using a functional programming language like Frege.

Acknowledgements

I would like to thank Prof. Dierk König for the ongoing support of this master thesis and the previous projects that built up the knowhow and confidence to tackle the problems covered in this thesis. This master's degree opened me the door to unknown territories and this thesis gave me the possibility to dive further into the world of functional programming in the context of Frege. Our meetings provided valuable opportunities to discuss issues and insights and get different perspectives on some of the ideas and obstacles encountered in this project. This exchange and the knowhow gained in our talks proved valuable guidance while working on the topics of this thesis, for which I am truly thankful for.

Bibliography

[1] "Frege Language Documentation," [Online]. Available: https://github.com/Frege/frege/blob/master/README.md. [Accessed 6 August 2019].

[2] "Wikipedia - Pure Function," [Online]. Available: https://en.wikipedia.org/wiki/Pure_function. [Accessed 5 September 2019].

[3] J. M. Christopher Allen, Haskell Programming from first principle, 2019.

[4] I. Wechsung, "Frege Language Specification," 14 May 2014. [Online]. Available: http://web.mit.edu/frege-lang_v3.24/Language.pdf.

[5] "Oracle Docs - Callable," [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html. [Accessed 9 August 2019].

[6] "Stackoverflow - Currying," [Online]. Available: https://stackoverflow.com/questions/36314/what-is-currying. [Accessed 14 August 2019].

[7] "Haskell Wiki - Partial Application," [Online]. Available: https://wiki.haskell.org/Partial_application. [Accessed 14 August 2019].

[8] "Frege Wiki - Calling Frege From Java," [Online]. Available: https://github.com/Frege/frege/wiki/Calling-Frege-from-Java-(from-release-3.24-on).

[9] "Jaceklaskowski - Mastering Apache Spark," [Online]. Available: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-rdd.html. [Accessed 13 August 2019].

[10] "Medium Blog Post on Apache Sparks Transformations," [Online]. Available: https://medium.com/@aristo_alex/how-apache-sparks-transformations-and-action-works-ceb0d03b00d0. [Accessed 8 August 2019].

[11] "Apache Spark RDD Documentation - Transformations," [Online]. Available: https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations. [Accessed 8 August 2019].

[12] "Apache Spark Documentation - SQL Programming Guide," [Online]. Available: https://spark.apache.org/docs/latest/sql-programming-guide.html. [Accessed 13 August 2019].

[13] "Apache Spark Examples," [Online]. Available: https://spark.apache.org/examples.html. [Accessed 8 August 2019].

[14] "Apache Spark Documentation - Cluster Overview," [Online]. Available: https://spark.apache.org/docs/latest/cluster-overview.html. [Accessed 8 August 2019].

[15] "FregIDE documentation," [Online]. Available: https://github.com/Frege/eclipse-plugin/wiki/fregIDE-Tutorial. [Accessed 13 August 2019].

[16] "Apache Spark Documentation - JavaSparkContext," [Online]. Available: https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/api/java/JavaSparkContext.html. [Accessed 26 June 2019].

[17] "Apache Spark Documentation - JavaRDD," [Online]. Available: https://spark.apache.org/docs/2.2.0/api/java/index.html?org/apache/spark/api/java/JavaRDD.html. [Accessed 16 August 2019].

[18] "StackOverflow - Serialization Debugger," [Online]. Available: https://stackoverflow.com/questions/22592811/task-not-serializable-java-io-notserializableexception-when-calling-function-ou. [Accessed 9 September 2019].

[19] "Vogella - Serialization," [Online]. Available: https://www.vogella.com/tutorials/JavaSerialization/article.html. [Accessed 9 August 2019].

[20] "Baeldung - Java Marker Interfaces," [Online]. Available: https://www.baeldung.com/java-marker-interfaces. [Accessed 14 August 2019].

[21] "StackOverflow - Spark Serialization," [Online]. Available: https://stackoverflow.com/questions/40596871/how-spark-handles-object. [Accessed 9 August 2019].

[22] "DZone Blog Post - Serialize Lambdas," [Online]. Available: https://dzone.com/articles/how-and-why-to-serialialize-lambdas. [Accessed 8 August 2019].

[23] "Stackoverflow - Serializable and Externalizable," [Online]. Available: https://stackoverflow.com/questions/817853/what-is-the-difference-between-serializable-and-externalizable-in-java. [Accessed 8 August 2019].

[24] "DevInsider - Compiler und Interpreter," [Online]. Available: https://www.dev-insider.de/der-unterschied-von-compiler-und-interpreter-a-742282/. [Accessed 5 September 2019].

[25] "Github - Frege Interpreter," [Online]. Available: https://github.com/Frege/frege-interpreter. [Accessed 13 August 2019].

[26] "Wikipedia - JSR," [Online]. Available: https://de.wikipedia.org/wiki/Java_Specification_Request. [Accessed 5 September 2019].

[27] "Oracle Java Docs - Scripting," [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/. [Accessed 5 September 2019].

[28] "Narkive Mailinglist Archive - Frege Scripting," [Online]. Available: https://frege-programming-language.narkive.com/dL3GXTkX/calling-frege-from-java-but-as-a-script-like-groovy. [Accessed 20 June 2019].

[29] "Apache Spark Documentation - RDD," [Online]. Available: https://spark.apache.org/docs/latest/rdd-programming-guide.html#resilient-distributed-datasets-rdds. [Accessed 8 August 2019].

Appendix

Installation guide

To setup the project on your local machine, please follow the steps in this installation guide. For additional information refer to the chapter “Setup”.

Install Eclipse

Follow the corresponding guides and tutorials in the Frege Wiki to set up the environment.
https://github.com/Frege/frege/wiki/Getting-Started
https://github.com/Frege/eclipse-plugin/wiki/fregIDE-Tutorial

Install FregIDE Eclipse plugin

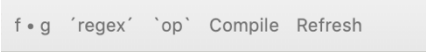
Follow the Installation tab in the FregIDE-tutorial (https://github.com/Frege/eclipse-plugin/wiki/fregIDE-Tutorial#installation)

Import project sources

Import the project sources by cloning and importing the git repository https://github.com/elrocqe/frege_spark.git as a project into the Eclipse IDE.

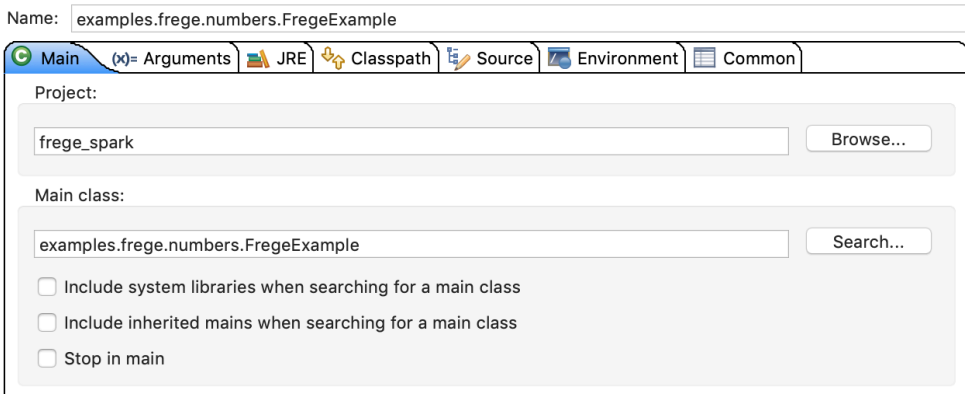
Enable Frege Builder

Right-click on the frege_spark project in one of the Explorer Views in Eclipse (Package or Project Explorer) to open the context menu and select “Enable Frege Builder”. Now your IDE should provide further toolbar entries when opening *.fr-files.



Create Run Configuration in Eclipse

Open the Run Configurations menu, which can be found through the Run Menu or by clicking on the arrow next to this icon. 
Create a new Run Configuration for the application you would like to run. Possible applications can be found in src/main/frege/examples and src/main/java/examples.



Run the application

Lastly, run the created RunConfiguration and check the console for output.

Instabilities and Troubleshooting

During development, issues and instabilities occurred and impacted the project and are documented here for further reference.

Issues with fregec.jar

It has to be noted that different parts of this setup use the fregec.jar, the command line compiler for the Frege language, and it is crucial to provide the same version to all components.
The IDE uses a fregec.jar, and it is also passed to Apache Spark. At the time of writing, the version most compatible was 3.24.405.

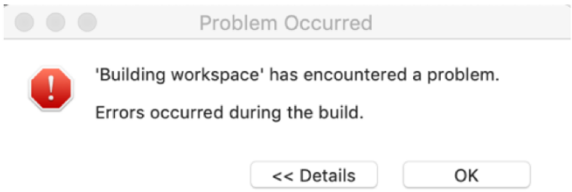
When running Frege's ScriptingEngine, there seems to be instabilities with which versions of the fregec.jar it is compatible. Below is the stack trace of the errors thrown for many different versions of fregec.jar. Among which were 3.25.84, which is the latest version at the time of writing, or 3.25.42, which is the version provided by Frege's Eclipse-plugin.

```
Exception in thread "main" java.lang.NoSuchMethodError:
frege.prelude.PreludeBase.maybe(Lfrege/run8/Lazy;Lfrege/run8/Func$U;Lfrege/prelude/PreludeBas
e$TMaybe;)Lfrege/run8/Lazy;
        at frege.scriptengine.FregeScriptEngine.lambda$null$25(FregeScriptEngine.java:1078)
        at frege.run8.Thunk.call(Thunk.java:230)
        at
frege.scriptengine.FregeScriptEngine.lambda$removeBindingVars$40(FregeScriptEngine.java:1137)
        at frege.prelude.PreludeBase$TST.lambda$performUnsafe$4(PreludeBase.java:10444)
        at frege.run8.Thunk.call(Thunk.java:230)
        at
frege.scriptengine.FregeScriptEngine$JFregeScriptEngine.eval(FregeScriptEngine.java:634)
        at javax.script.AbstractScriptEngine.eval(AbstractScriptEngine.java:264)
        at script.ScriptExecutor.loadFunctions(ScriptExecutor.java:69)
        at script.ScriptExecutor.main(ScriptExecutor.java:41)
```

The ScriptEngine is working as expected with version 3.24.405.

Eclipse Build Errors

The IDE works best with the default fregec.jar that is provided by the FregIDE plugin for Eclipse. At the time of writing this was version 3.25.42 (BundleVersion 3.24.366).
However, to make the interpreter work during development, the fregec.jar version 3.24.405 proved most stable.
It is possible to change the fregec.jar, but this might cause instabilities for certain versions.
To exchange the fregec.jar, simply download the desired version and put it in the folder where Eclipse links the corresponding referenced libraries (e.g. \Users\<username>\.p2\pool\plugins\frege.ide_3.25.42\lib).
But keep in mind that changing the fregec.jar may lead to instabilities with the IDE.
It occurred that sometimes, for example when the computer was rebooted, the sources were not compiled properly and the IDE indicated errors such as ClassNotFoundException.
In that case you can open the Frege file of the missing class and force compilation manually through the 'Compile'-Button in the IDE. This will usually resolve the issue. Do this for all the necessary files along the nested import hierarchy. Why this error occurs is unclear, but manually recompiling the files helps to work around the issue.
When exchanging the fregec.jar, it is also possible that the Eclipse IDE becomes instable and throws multiple errors such as the following:



At some points, these issues strongly hindered development, but other times they completely disappeared. It is unclear how to resolve these issues but exchanging the version of the fregec.jar and restarting the IDE helped alleviate the issues.

ScriptEngine Error

Even in the running version, there is an error thrown in correlation with the ScripingEngine.

```
java.lang.NoSuchMethodError:
frege.scriptengine.FregeScriptEngine.<init>(Ljavax/script/ScriptEngineFactory;)V
        at
frege.scriptengine.FregeScriptEngineFactory.getScriptEngine(FregeScriptEngineFactory.java:75)
        at javax.script.ScriptEngineManager.getEngineByName(ScriptEngineManager.java:238)
        at script.ScriptReader.executeFunctionFromJar(ScriptReader.java:72)
        at bindings.FunctionHelper$2.call(FunctionHelper.java:36)
        at
org.apache.spark.api.java.JavaPairRDD$$anonfun$toScalaFunction$1.apply(JavaPairRDD.scala:1040
)
        at scala.collection.Iterator$$anon$11.next(Iterator.scala:410)
        at scala.collection.Iterator$$anon$10.next(Iterator.scala:394)
        at scala.collection.Iterator$class.foreach(Iterator.scala:891)
        at scala.collection.AbstractIterator.foreach(Iterator.scala:1334)
        at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:59)
        at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:104)
        at scala.collection.mutable.ArrayBuffer.$plus$plus$eq(ArrayBuffer.scala:48)
        at scala.collection.TraversableOnce$class.to(TraversableOnce.scala:310)
        at scala.collection.AbstractIterator.to(Iterator.scala:1334)
        at scala.collection.TraversableOnce$class.toBuffer(TraversableOnce.scala:302)
        at scala.collection.AbstractIterator.toBuffer(Iterator.scala:1334)
        at scala.collection.TraversableOnce$class.toArray(TraversableOnce.scala:289)
        at scala.collection.AbstractIterator.toArray(Iterator.scala:1334)
        at org.apache.spark.rdd.RDD$$anonfun$take$1$$anonfun$29.apply(RDD.scala:1364)
        at org.apache.spark.rdd.RDD$$anonfun$take$1$$anonfun$29.apply(RDD.scala:1364)
        at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:2101)
        at org.apache.spark.SparkContext$$anonfun$runJob$5.apply(SparkContext.scala:2101)
        at org.apache.spark.scheduler.ResultTask.runTask(ResultTask.scala:90)
        at org.apache.spark.scheduler.Task.run(Task.scala:121)
        at org.apache.spark.executor.Executor$TaskRunner$$anonfun$10.apply(Executor.scala:402)
        at org.apache.spark.util.Utils$.tryWithSafeFinally(Utils.scala:1360)
        at org.apache.spark.executor.Executor$TaskRunner.run(Executor.scala:408)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)
```

The cause for this error was not investigated further and the error remains when running the applications that involve the ScriptEngine as it did not affect the execution of the applications.

Issues with frege-spark.jar

When running an application on Apache Spark it is the frege-spark.jar file that gets executed, on the worker nodes. Always make sure that all necessary files are packed into the JAR-file and that all the files are up-to-date.

Possible errors and exceptions include:

- NoSuchMethodError
  In case the FunctionPool provided in the frege-spark.jar does not include certain functions which might be added in the source code, this error will be thrown. Always make sure to run the latest version of the application.

- InvalidClassException
  The same as above also goes when adjusting Java files. Making adjustments in the Java helper classes but not updating the deployed frege-spark.jar will result in an InvalidClassException

```
19/08/29 14:00:24 WARN TaskSetManager: Lost task 0.0 in stage 2.0 (TID 2,
192.168.100.80, executor 0): java.io.InvalidClassException:
bindings.FunctionExperimentsHelper$6; local class incompatible: stream classdesc
serialVersionUID = 5226414315631083210, local class serialVersionUID =
631066412953032677
```

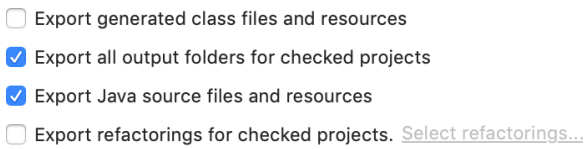
- NullPointerException
  Similar issues can also arise when calling a script which is not loaded into the interpreter on the node and is not available to be executed.

```
19/08/08 13:43:43 WARN TaskSetManager: Lost task 0.0 in stage 0.0 (TID 0,
192.168.0.171, executor 0): java.lang.NullPointerException
at script.ScriptExecutor.executeScriptFunction(ScriptExecutor.java:32)
at bindings.FunctionHelper$3.call(FunctionHelper.java:45)
```

- ClassNotFoundException
  Eclipse might not add all the necessary classes to the JAR by default, as the generated output from Java and Frege files is treated differently. This may result in an error such as the following:

```
java.lang.ClassNotFoundException:
examples.frege.numbers.TryoutIntegrationExampleFrege$NativeFunction$2
at
org.apache.spark.serializer.JavaDeserializationStream$$anon$1.resolveClass(JavaSeriali
zer.scala:67)
```

  In the case where the compilation of Frege files is missing, switching between the options “Export generated class files and resources” and “Export all output folders for checked projects” during the Eclipse Export process will make sure that all necessary files are packed into the jar-file.



Code Example

HelloWorld Example

```
module examples.frege.HelloWorld where

main :: [String] -> IO ()
main args = do
  result = 1.0 + 1.0
  println "Hello World"
  println result
```

snippets.HelloWorld.fr

```
/*
  Source code is in UTF-8 encoding. The following symbols may appear, among others:
  α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ φ χ ψ ω « • ¦ » ∀ ∃ :: … → ← fi fl ƒ
  If you can't read this, you're out of luck. This code was generated with the frege compiler version 3.25.42
  from src/main/frege/examples/frege/HelloWorld.fr Do not edit this file! Instead, edit the source file and
recompile.
*/

package examples.frege;

import frege.run8.Func;
import frege.run8.Lazy;
import frege.run8.Thunk;
import frege.run.Kind;
import frege.run.RunTM;
import frege.runtime.Meta;
import frege.runtime.Phantom.RealWorld;
import frege.Prelude;
import frege.control.Category;
import frege.control.Semigroupoid;
import frege.java.IO;
import frege.java.Lang;
import frege.java.Util;
import frege.java.util.Regex;
import frege.prelude.Maybe;
import frege.prelude.PreludeArrays;
import frege.prelude.PreludeBase;
import frege.prelude.PreludeDecimal;
import frege.prelude.PreludeIO;
import frege.prelude.PreludeList;
import frege.prelude.PreludeMonad;
import frege.prelude.PreludeText;

@SuppressWarnings("unused")
@Meta.FregePackage(
  source="src/main/frege/examples/frege/HelloWorld.fr", time=1558534495398L, jmajor=1, jminor=8,
  imps={
    "frege.Prelude", "frege.prelude.PreludeArrays", "frege.prelude.PreludeBase",
"frege.prelude.PreludeDecimal",
    "frege.prelude.PreludeIO", "frege.prelude.PreludeList", "frege.prelude.PreludeMonad",
"frege.prelude.PreludeText",
    "frege.java.util.Regex"
  },
  nmss={
    "Prelude", "PreludeArrays", "PreludeBase", "PreludeDecimal", "PreludeIO", "PreludeList", "PreludeMonad",
    "PreludeText", "Regexp"
  },
```

```
 symas={}, symcs={}, symis={}, symts={},
 symvs={
  @Meta.SymV(
   offset=40, name=@Meta.QName(pack="examples.frege.HelloWorld", base="main"), stri="s(u)",
   sig=1, depth=1, rkind=13
  )
 },
 symls={},
 taus={
  @Meta.Tau(kind=2, suba=0, tcon={@Meta.QName(kind=0, pack="frege.prelude.PreludeBase",
base="[]")}),
  @Meta.Tau(kind=2, suba=0, tcon={@Meta.QName(kind=0, pack="frege.prelude.PreludeBase",
base="StringJ")}),
  @Meta.Tau(kind=2, suba=0, tcon={@Meta.QName(kind=0, pack="frege.prelude.PreludeBase",
base="Char")}),
  @Meta.Tau(kind=0, suba=1, subb=2), @Meta.Tau(kind=0, suba=0, subb=3),
  @Meta.Tau(kind=2, suba=0, tcon={@Meta.QName(kind=0, pack="frege.prelude.PreludeBase",
base="ST")}),
  @Meta.Tau(kind=2, suba=0, tcon={@Meta.QName(kind=0, pack="frege.prelude.PreludeBase",
base="RealWorld")}),
  @Meta.Tau(kind=0, suba=5, subb=6),
  @Meta.Tau(kind=2, suba=0, tcon={@Meta.QName(kind=0, pack="frege.prelude.PreludeBase",
base="()")}),
  @Meta.Tau(kind=0, suba=7, subb=8)
 },
 rhos={@Meta.Rho(rhofun=false, rhotau=4), @Meta.Rho(rhofun=false, rhotau=9), @Meta.Rho(sigma=0,
rhotau=1)},
 sigmas={@Meta.Sigma(rho=0), @Meta.Sigma(rho=2)}, exprs={@Meta.Expr()}
)
final public class HelloWorld  {

final public static Func.U<RealWorld, Short> $main(final Lazy<PreludeBase.TList<String/*<Character>*/>>
arg$1) {
 return PreludeMonad.IMonad_ST.<RealWorld, Short, Short>$gt$gt(
         Prelude.<String/*<Character>*/>println(PreludeText.IShow_String.it, "Hello World"),
         Thunk.<Func.U<RealWorld, Short>>shared(
             (Lazy<Func.U<RealWorld, Short>>)(() -> Prelude.<Double>println(PreludeText.IShow_Double.it,
1.0 + 1.0D))
            )
        );
}

 public static void main(final java.lang.String[] argv) {
  try {
   frege.run.RunTM.argv = argv;

    PreludeBase.TST.<Short>performUnsafe($main
         (Thunk.lazy(PreludeArrays.IListSource_JArray.<String/*<Character>*/>toList(argv))).call()
      ).call();
   frege.runtime.Runtime.stdout.get().close();
   frege.runtime.Runtime.stderr.get().close();

  } finally { frege.run.Concurrent.shutDownIfExists(); }
 }
}
```

snippets.HelloWorld.java (compiled output of snippets.HelloWorld.fr)

Corresponding HelloWorld.class-file omitted as it consists of Java Bytecode.

StackTrace - TaskNotSerializable

```
Exception in thread "main" org.apache.spark.SparkException: Task not serializable
at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:403)
at
org.apache.spark.util.ClosureCleaner$.org$apache$spark$util$ClosureCleaner$$clean(ClosureClea
ner.scala:393)
at org.apache.spark.util.ClosureCleaner$.clean(ClosureCleaner.scala:162)
at org.apache.spark.SparkContext.clean(SparkContext.scala:2326)
at org.apache.spark.rdd.RDD$$anonfun$map$1.apply(RDD.scala:371)
at org.apache.spark.rdd.RDD$$anonfun$map$1.apply(RDD.scala:370)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:151)
at org.apache.spark.rdd.RDDOperationScope$.withScope(RDDOperationScope.scala:112)
at org.apache.spark.rdd.RDD.withScope(RDD.scala:363)
at org.apache.spark.rdd.RDD.map(RDD.scala:370)
at org.apache.spark.api.java.JavaRDDLike$class.map(JavaRDDLike.scala:93)
at org.apache.spark.api.java.AbstractJavaRDDLike.map(JavaRDDLike.scala:45)
at spark.bindings.JavaRDD$TJavaRDD.lambda$5(JavaRDD.java:233)
at frege.prelude.PreludeBase$TST.lambda$$gt$gt$eq$3(PreludeBase.java:10439)
at frege.prelude.PreludeBase$TST.lambda$$gt$gt$eq$2(PreludeBase.java:10442)
at frege.run8.Thunk.call(Thunk.java:230)
at examples.frege.numbers.IntegrationExampleFrege.main(IntegrationExampleFrege.java:906)
Caused by: java.io.NotSerializableException:
examples.frege.numbers.IntegrationExampleFrege$$Lambda$38/1209962934
Serialization stack:
- object not serializable (class:
examples.frege.numbers.IntegrationExampleFrege$$Lambda$38/1209962934, value:
examples.frege.numbers.IntegrationExampleFrege$$Lambda$38/1209962934@6cd56321)
- field (class: bindings.Functions$3, name: val$fStored, type: interface frege.run8.Func$U)
- object (class bindings.Functions$3, bindings.Functions$3@6f139fc9)
- field (class: org.apache.spark.api.java.JavaPairRDD$$anonfun$toScalaFunction$1, name:
fun$1, type: interface org.apache.spark.api.java.function.Function)
- object (class org.apache.spark.api.java.JavaPairRDD$$anonfun$toScalaFunction$1,
<function1>)
at
org.apache.spark.serializer.SerializationDebugger$.improveException(SerializationDebugger.sca
la:40)
at org.apache.spark.serializer.JavaSerializationStream.writeObject(JavaSerializer.scala:46)
at org.apache.spark.serializer.JavaSerializerInstance.serialize(JavaSerializer.scala:100)
at org.apache.spark.util.ClosureCleaner$.ensureSerializable(ClosureCleaner.scala:400)
... 16 more
```