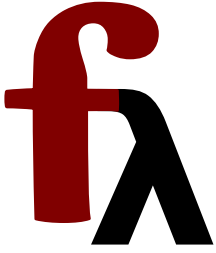
Getting started with Frege

Lambda Days 2016

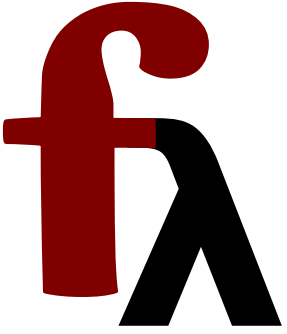


lambda
DAλS

Lech Głowiak

Frege Programming Language

JVM language in spirit of Haskell



Haskell for the JVM

https://github.com/Frege/frege

Features

- Purely functional
- Higher order functions
- Non-strict evaluation (default)
- Immutable values
- Algebraic data types
- Strongly, statically typed
- Type inference
- Terse syntax
- *Runs on JVM*
- *Interoperability with JVM languages*

Distribution

Prerequisites: JDK 7 or 8
Releases: https://github.com/Frege/frege/releases or Maven/Sonatype

Make aliases in environment and you are set up for command line work!

```
alias frege='java -Xss1m -cp build:$FREGE_HOME/fregec.jar'
alias fregec='java -Xss1m -jar $FREGE_HOME/fregec.jar -d build'
```

Wiki: Getting started

Read Evaluate Process Loop

Download and run

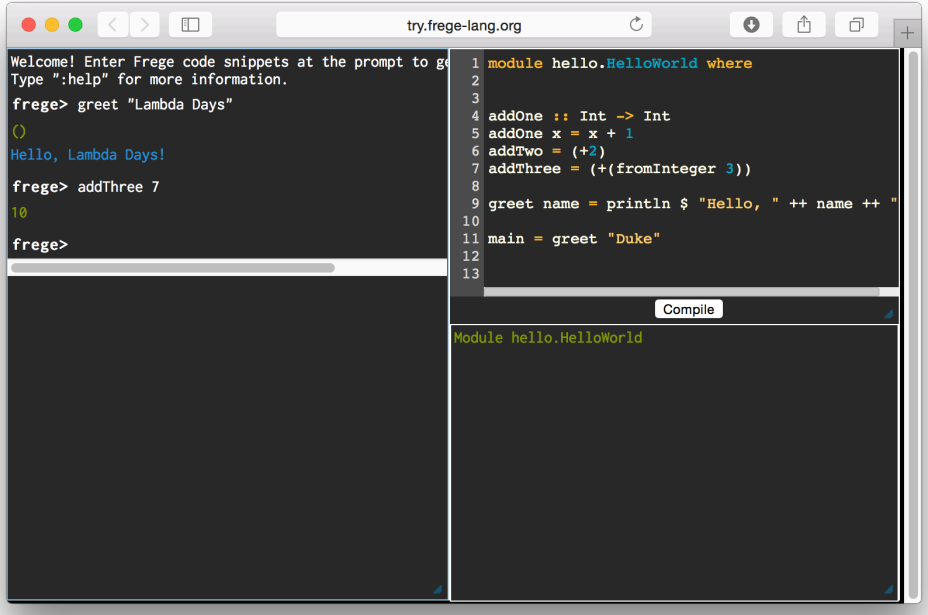
```
wget https://github.com/Frege/frege-repl/releases/\
download/1.4-SNAPSHOT/frege-repl-1.4-SNAPSHOT.zip &&
unzip frege-repl-1.4-SNAPSHOT.zip &&
./frege-repl-1.4-SNAPSHOT/bin/frege-repl
```

Build your own

```
git clone https://github.com/Frege/frege-repl &&
cd frege-repl &&
./gradlew
```

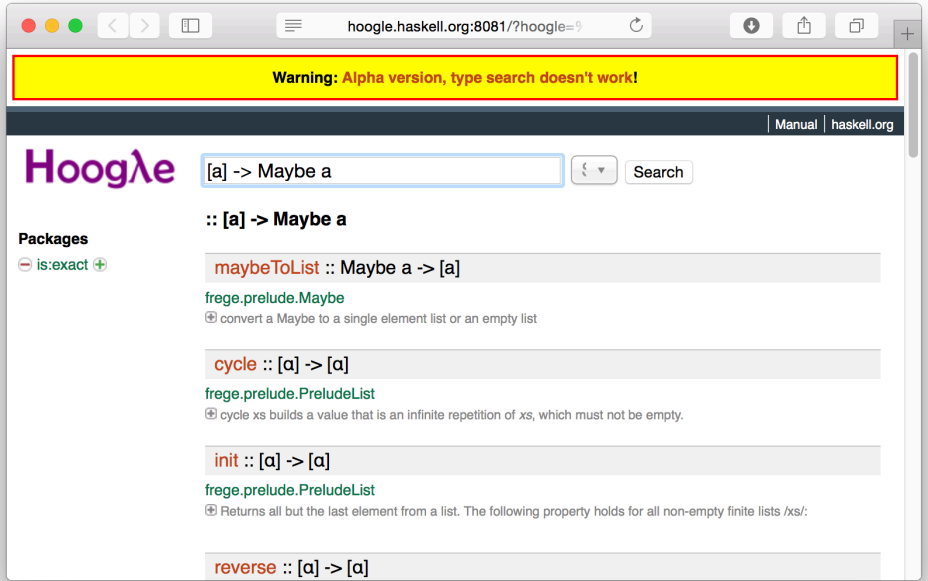
Online REPL

There is at least one online REPL: http://try.frege-lang.org



Hoogle for Frege (formerly Froogle)

Hoogle 5 for Frege: http://hoogle.haskell.org:8081



Parsing JSON - first attempt

Using build-in JSON module

```
import Data.JSON

type ErrorStr = String
type Validate = String -> Either ErrorStr String
-- Parsing to JSON AST is enough for validation
parseToValue :: String -> Either ErrorStr JSON.Value
parseToValue = JSON.parseJSON

validateJson :: Validate
validateJson a = case (parseToValue a) of
                   Left err -> Left err
                   Right v -> Right (show v)
```

Second attempt: using 3rd party JVM library

To call JVM code in Frege use `native` declarations:

- declare what classes and methods you want to use
- specify if instances are immutable
- what is *purity* level of functions: `pure`, `ST`, `IO`
- specify if functions throw or return `null`

Second attempt: using 3rd party JVM library

- Java code to simplify `native` declarations
- `parse` is `pure`

```
private static ObjectReader r = new ObjectMapper().reader();

public static JsonNode parse(String s)
        throws IllegalArgumentException{
    try {
        return r.readTree(s);
    }catch(JsonProcessingException e){
        throw new IllegalArgumentException(e.getMessage());
    }catch(Exception e){
        throw new IllegalArgumentException(e.toString());
    }
}
```

*because I'm not so good in Haskell

Second attempt: using 3rd party JVM library

```
data JacksonJsonNode =
  pure native com.fasterxml.jackson.databind.JsonNode where
   pure native toString :: JacksonJsonNode -> String

pure native getMessage :: IllegalArgumentException -> String

data JacksonWrapper =
  native com.github.LGLO.JacksonWrapper where
    pure native parse com.github.LGLO.JacksonWrapper.parse ::
      String -> (IllegalArgumentException | JacksonJsonNode)

validateJson :: Validate
validateJson a = case (JacksonWrapper.parse a) of
                   Left ex -> Left (ex.getMessage)
                   Right j -> Right j.toString
```

Native declarations

`pure native` for immutable objects only

```
data Rational = pure native immutable.Rational where
    -- static
    pure native new :: Long -> Long -> Rational
    pure native integer immutable.Rational.integer :: Long ->
                                                      Rational
    -- instance
    pure native toString :: Rational -> String
    pure native add :: Rational -> Rational -> Rational
    pure native div :: Rational -> Rational ->
                       (IllegalArgumentException | Rational)
    pure native divNull :: Rational -> Rational -> Maybe Rational
```

- native functions are strict on all arguments
- native functions type must not have type class constraints

Native declarations

`native` for mutable objects

```
-- AB is an envelope for final int 'a' and non final int 'b'
data AB = native mutable.AB where
    native new :: Int -> Int -> ST s (Mutable s AB)
    pure native a getA :: AB -> Int
    native getB :: Mutable s AB -> ST s Int
    native setB :: Mutable s AB -> Int -> ST s ()
```

`Mutable` cannot be instantiated by nor passed to `pure` function - it is compiler defense.

```
readonly :: (m -> b) -> Mutable s m -> ST s b

freeze :: Freezable f => Mutable s m -> ST s m
thaw :: Freezable f => m -> ST s (Mutable s m)
```

Native declarations

`mutable native`:

- does not allow `STMutable s A` return
- restricts return type to `ST RealWorld`, that is `IO`
- you don't have to type `Mutable RealWorld` everywhere

```
data TT = mutable native io.TelephoneTime where
    native new :: () -> IO TT -- IO = ST RealWorld
    native printTime :: TT -> IO ()

data TT2 = native io.TelephoneTime where
    native new :: () -> STMutable RealWorld TT2
    native printTime :: Mutable RealWorld TT2 -> IO ()
```

native-gen-tool

- Tool that helps writing `native` declarations
- Rather to be run once, not every build
- Workflow:
  - Edit 'types.properties' to mark classes as `pure`, `st` or `io`
  - Run `frege.nativegen.Main` with appropriate options
  - Cut interesting parts from generated source and work on it:
    - Wrap return type in `Maybe` or `Either` for nulls and exceptions
    - Fix some other return type issues
    - Mark pure methods as `pure`
    - Delete what doesn't compiles and is not essential ;)

```
//types.properties
java.math.BigInteger=pure,Integer
java.nio.ByteBuffer=st
java.nio.channels.FileChannel=io
```

https://github.com/Frege/frege-native-gen

native-gen-tool

Frege code took care of what was done in wrapper in earlier example.

```
data ObjectReader =
  pure native com.fasterxml.jackson.databind.ObjectReader where
    pure native readTree :: ObjectReader -> String ->
                                          (IOException | JsonNode)

data ObjectMapper =
  native com.fasterxml.jackson.databind.ObjectMapper where
    native new :: () -> STMutable s ObjectMapper
    native reader :: Mutable s ObjectMapper -> ST s ObjectReader
```

```
mkReader = ST.run (
    do mutableMapper <- ObjectMapper.new () -- tainted with Mutable
       -- returns without Mutable because ObjectReader is immutable
       mutableMapper.reader
    )

validateJson a = case (mkReader.readTree a) of
                   Left ex -> Left ex.getMessage
                   Right jsonNode -> Right jsonNode.toString
```

Webservice #1 - Java calls Frege

Java `main` runs **SparkJava**. **SparkJava** handlers call **Frege** code.

```
private static Route validateHandler = (req, res) -> {
  String body = req.body();
  //For Frege types it is safe to assume result is lazy*.
  TEither result = Delayed.<TEither>forced(validateJson(body));
  //It is safe to always pass strict arguments*.
  res.status(statusCode(result));
  return output(result);
};
```

In **Frege 3.23** there is runtime type check for instance of `Delayed`.

```
validateJson :: Validate
validateJson a = case (parseToValue a) of Left err -> Left err
                                          Right v -> Right (show v)
statusCode :: Either ErrorStr String -> Int
statusCode = either (const 400) (const 200)

output :: Either ErrorStr String -> String
output = either id id
```

SparkJava: http://sparkjava.com

Webservice #2 - Frege only* solution

Chinook for web
Sirocco for DB

```
module jsonlint.App where

import chinook.Chinook (get)
import jsonlint.DB
import jsonlint.Handlers (validateHandler)

main _  = do
  DB.createRequestsTable
--get  :: String -> (IO Request -> IO Response) -> IO ()
  get    "/validate"       validateHandler
  get    "/"               validateHandler
```

https://github.com/januslynd/chinook
https://github.com/januslynd/sirocco

Webservice #2 - Frege only* solution

***Chinook - JavaSpark* in Frege**

```
validateHandler :: IO Request -> IO Response
validateHandler req = do body <- req.body
                         validate body

validate :: Maybe String -> IO Response
validate Nothing = missingBody
validate (Just "") = missingBody
validate (Just body) =
  let result = validateJson body
    in do
         _ <- DB.logRequest body (isValid result)
         return (toResponse result)

missingBody = IO.return (mkResponse 400 "There is no BODY.")

toResponse :: Either String String -> Response
toResponse = either (mkResponse 400) (mkResponse 200)

mkResponse status output =
    response.{ status = status, output = Just output }
```

Webservice #2 - Frege only* solution

***Sirocco* DB access**

```
connection :: IO Connection
connection = createConnection databaseURI "sa" ""

logRequest :: String -> Bool -> IO Int
logRequest text valid =
  let sql = Just "INSERT INTO requests (input, valid) values (?, ?)"
      params = [toParam (Just text), toParam (Just valid)]
        in do update connection sql params

createRequestsTable :: IO ()
createRequestsTable = do
--execute :: IO Connection -> Maybe String -> IO Bool
  _ <- execute connection (Just "DROP TABLE requests")
  stmt <- execute connection (Just createTableSQL)
  println $ "Table 'requests' created: " ++ (show stmt)
```

Gradle

https://github.com/Frege/frege-gradle-plugin

- Compiles Java before Frege by default
- Use sub-projects if you need to call Frege code from Java
- `-x test` if you have no QuickCheck test

sbt

https://github.com/earldouglas/sbt-frege

- Compiles Frege first by default
- Use sub-projects if you need to use native declarations

Leiningen

https://github.com/Frege/frege-lein-plugin

- Compiles Java before Frege
- Use frege-lein-template for quick start

Eclipse plugin

https://github.com/Frege/eclipse-plugin
Rumor says it's great, especially when you hack on Frege itself.

Intellij IDEA misses good quality plugin

But you can upvote it on: https://youtrack.jetbrains.com/issue/IDEABKL-7108

Community

- Gitter: https://gitter.im/Frege/frege
- IRC: **#frege** at https://freenode.net (connected to Gitter via bot)
- Mailing group: http://groups.google.com/group/frege-programming-language
- Stackoverflow: **frege** tag

Contributors are welcome - **Frege** is a nice thing to hack on!

Links

- "Practical type inference for arbitrary-ranked types" by Simon Peyton Jones http://research.microsoft.com/en-us/um/people/simonpj/papers/higher-rank/putting.pdf
- "Learn You a Haskell for Great Good!" by Miran Lipovača http://learnyouahaskell.com
- "Haskell 2010 Language Report" http://www.haskell.org/onlinereport/haskell2010/
- Presented code + plugins examples + quite more [https://github.com/LGLO/LambdaDays2016]

Questions???

Lech Głowiak

> scalac

@LGLO

@LechGlowiak

Thank you!

I kindly ask for **frank** feedback!

All presented code is available at
[https://github.com/LGLO/LambdaDays2016]
+ plugins examples + everything that was presented on slides