Frege,

a non-strict, pure functional programming language for the JVM

presented by

Ingo Wechsung

Named after G. Frege, who had a concept of higher order functions

Wie nun Funktionen von Gegenständen grundverschieden sind, so sind auch Funktionen, deren Argumente Funktionen sind und sein müssen, grundverschieden von Funktionen, deren Argumente Gegenstände sind und nichts anderes sein können. Diese nenne ich Funktionen erster, jene Funktionen zweiter Stufe.

G. Frege, “Funktion und Begriff”, 1891

Gottlob Frege

* 8. November 1848; † 26. July 1925

German mathematician, logician and philosopher. He is considered to be one of the founders of modern logic and made major contributions to the foundations of mathematics.

Frege lived and worked in Jena.

I recommend: Die Grundlagen der Arithmetik, eine logisch mathematische Untersuchung über den Begriff der Zahl.

In the spirit of Haskell ...

- Haskell compatible syntax (well, almost)
- strong static type system (Hindley-Milner + higher rank types + type classes)
- non-strict/lazy evaluation guided by strictness analysis
- pure functional ...
  - native interface for using Java methods and classes
  - input/output and mutable values are encapsulated in inescapable monads
  - many data types, standard functions and modules known from Haskell 2010

... but still quite different

- primitive types are borrowed from Java, as well as String
- almost no runtime system, "primitives" are just standard Java methods or operators
- no general foreign function interface, instead specialisation on JVM
    - can't use C-ish stdio like in Haskell
    - whenever possible, use Java SE API
- different concurrency model (*green threads*)
- different type class hierarchy. (We'll soon be fully "semigroupoidal".)

Compiler & Language facts

- Compiler written in … Frege, of course!
- Compiler generates Java source code
  - Maybe the only JVM language that does this?
- The language has so called *documentation comments*. They make it down to the class file and are used by the eclipse plugin and the documentation tool.

WTF did you not just write a GHC backend for the JVM?

- It's not easily possible, see GHC FAQ Why isn't GHC available for .NET or on the JVM
- Historical reasons:
  - wrote some perl code that implemented type inference.
  - added a parser
  - added code generation (to perl)
  - decided it's too slow, took the opportunity to learn Java and started over in Java
  - ambition: It must be possible to become self-hosting!
  - the result was horrible, but good enough to write Frege3
  - thought this could be useful for others and went public
- Had I just idled, nobody would blame me today for not having done a GHC backend.

Target Audience, Motivation

- Java programmers: perceive writing Frege code as just another and more fun way to write Java code. Not very successful until now.
- Haskell programmers: use Frege as substitute for the missing JVM-Haskell.
  - I claim without too much exaggeration that Frege is not only on par with Haskell 2010. It even has some extra goodies. (yet, missing libraries.)
  - That being said, compared to GHC, it is still a "poor man's Haskell"
- Interestingly, some users/contributors are Scala convertites.

JVM ecosystem

We enjoy a rich choice of functional languages or languages with functional influence on the JVM:

- Scala (strict, strongly typed, not pure)
- Clojure (strict, dynamic, not pure)
- Yeti and other ML dialects (strict, strongly typed, not pure)
- CAL, an earlier attempt to make a Haskell-like language, by Business Objects, now SAP (non strict, strongly typed, not pure)
- Ruby, Python
- the “better Java”s: Groovy, Kotlin, Ceylon, ...

Laziness Pro & Contra

- Pro:
  allow for infinite data structures, easier algorithms
  save unneeded work (even in traditional languages)
- Contra:
  it comes at a cost
  unpredictability of memory usage
- in Frege:
  No design choice; must have Haskell semantics!
  As it is there anyway, used for solving the tail call problem (see Example even/odd)
  bang patterns indicate strictness, strictness analyzer detects whole lot of strictness

Purity Pro & Contra (not!)

- Everything that can be said about the issue has been said already.
- If you don't like the strongly statically typed, lazy, pure paradigm, then neither Haskell nor Frege is for you.
- That being said, laziness presupposes purity to a certain degree:

```
let haha = (print “ha”, print “ho”)
in (haha, print “hi”)
```

In a hypothetical lazy, non-pure language, what should be the meaning?

Strong Static Type System Pro & Contra

from Cartesian Closed Comics:

(picture removed to avoid trouble rgd. copyrights, etc., please look it up yourself)

What are higher ranked types?

- often confused with higher *kinded* types
- the problem with Hindley-Milner type inference:
  - we can write higher order functions, but the functions we get passed are *monomorphic*:

    ```
    map :: forall a b.(a -> b) -> [a] -> [b]
    ```

  - When used, a possibly polymorphic function gets *instantiated* at a specific type

    ```
    show :: forall s.Show s => s -> String
    map show [1,2,3]  -- show :: Int -> String
    ```

  We can't write a function that takes a polymorphic list transformation function like `tail`, `sort`, `reverse`, etc. and applies it to <u>differently</u> typed lists.

Undecidability of HR type inference

- HM has the nice property that it can infer the most general (principal) type for any expression.
  - The restriction being that function arguments are assumed to be monomorphic.
  - We can't drop that assumption without destroying type inference altogether. Consider

    ```
    both f xs ys = (f xs, f ys)
    ```

    which would get a type like:

    ```
    (forall a b.a -> b) -> c -> d -> (e, f)   -- ???
    ```

    This is just unusable!

Type Annotations to the rescue!

- higher rank functions must be annotated
  - at least one argument has a **forall**
- type inference will work for the rest of the program
- not a big deal, as the tendency is to annotate top level functions anyway for documentation purpose
- same thing in GHC-Haskell with `RankNTypes`
- actually needed in the ST Monad (rank-2-type)

No assignments, no side effects, no flow of control ...

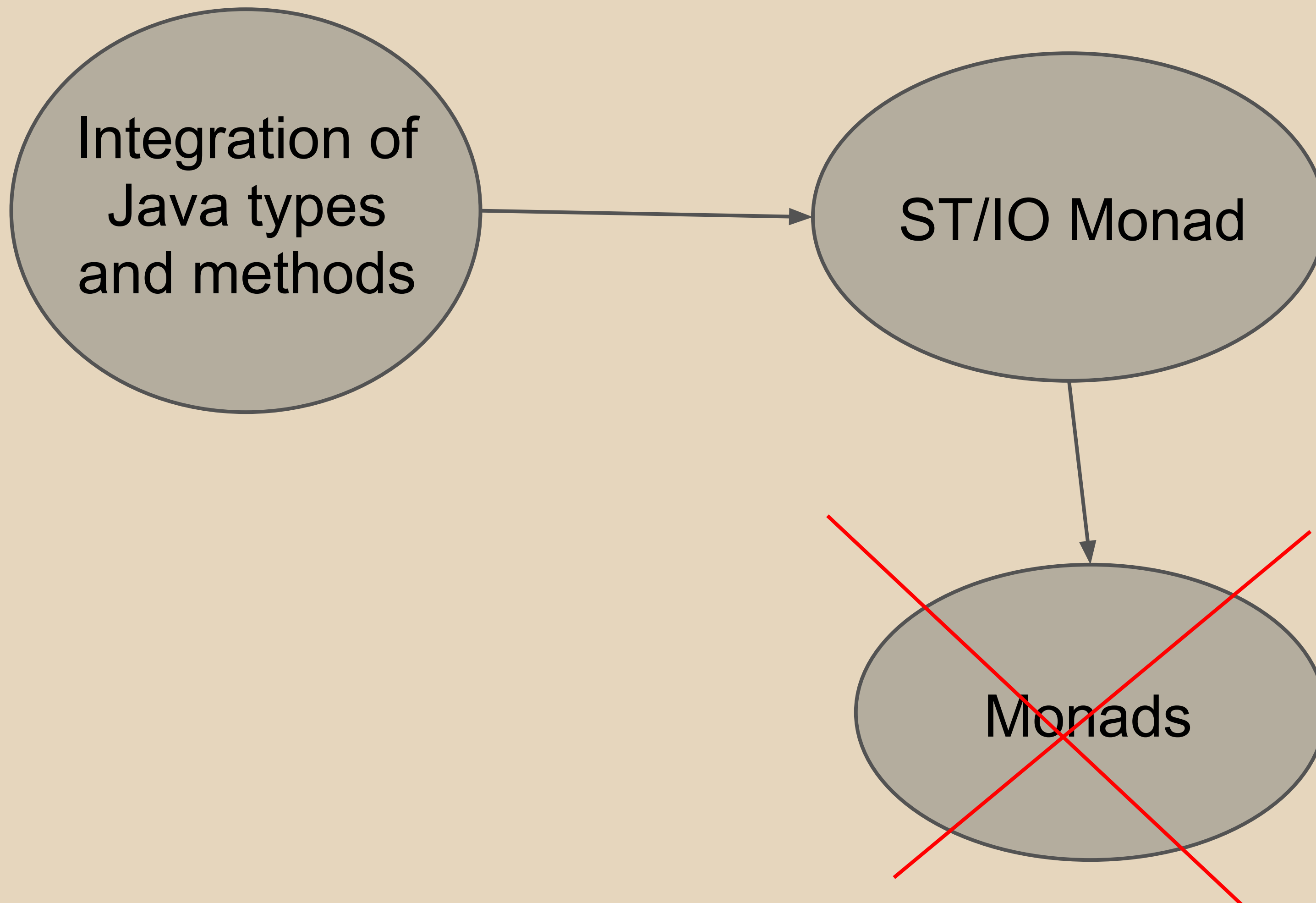
The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life, in the hope that it will make him virtuous.

John Hughes “Why FP Matters”, 1990

In short, Haskell is the world's finest imperative programming language.

Simon Peyton-Jones “Tackling the Awkward Squad”, 2010

I am trying the impossible ...



The ST Type
How to deal with side effects

- Implementation of the idea described in the paper Lazy Functional State Threads by Simon Peyton-Jones.
  - A value of type `ST s a` is a state transformer, where *s* symbolizes the mutable state and *a* is the result.
  - State transformers can be combined in the usual (monadic) way, giving new state transformers.
  - Some of them can be run to get a final pure value.
  - As long as all mutable data have the phantom type `s` in their type, the type system can ensure that they cannot escape the impure realm:

    ```
    run :: (forall ß.ST ß a) -> a
    ```

  - encapsulation of unobservable temporary effects

IO actions are ST actions

```
type IO = ST RealWorld
```

- A value of type `IO a` is an action that, when executed, may perform some <u>observable</u> change in the real world and computes a result of type `a`.
- Unlike polymorphic ST actions, cannot be run to get a pure value.
- Can be combined with other IO actions and ST actions that are polymorphic in the state.

IO/ST actions as first class values

```
some = [print “hi”, launchRockets, print “ho”]
more = (print “foo”, print “bar”)
hifoo = do { head some; fst more }
main ::  [String] -> IO ()
```

- no rockets launched here!
- only when `hifoo` is executed, get the actions executed
- for this, `hifoo` must be part of another action that gets executed, which must be part of yet another action that gets executed, which must be part of yet another action that gets executed, which must be part of yet another action that gets executed, which must be part of yet another action that gets executed
- `main`, applied to the list of command arguments, is the only exception: it gets executed by the runtime system.
  - and there is “*the function whose name must not be mentioned*”

The challenge in interfacing Java

- In Java, we have:
  - immutable data (rarely)
  - mutable data
  - data that are technically mutable, but are actually used in a pure fashion by some set of methods.
  - pure methods on immutable data
  - impure methods on immutable data
  - pure methods on mutable data
  - impure methods on mutable data
  - any method may return `null`
- In the general case, we don’t know which is which. We must trust the programmer. We want to guide him through this mess.

How to use Java Types from Frege

- <u>immutable data</u>
  ```
  data Bool = pure native boolean
  data Exc  = pure native java.lang.Exception
  ```
  - can be used like any other type, in fact, all primitive types and `String` are defined this way.
- <u>mutable data</u>
  ```
  data SB = native java.lang.StringBuilder
  ```
  - Compiler makes sure that in native functions only `Mutable s SB` is used.
- <u>data used with IO activities only</u>
  ```
  data Rdr = mutable native java.io.Reader
  ```
  essentialy saves typing `Mutable RealWorld Rdr` all the time

Observations rgd. mutable values

- Can only come into existence through java methods.
- Can only be mutated through java methods.
- Once the type of such methods is given honestly, it will infect the rest of the program with ST, IO and state phantom types (examples follow later). This is meant as a help for the disciplined ~~medieval monk~~ programmer.
- Still, it will be possible to run pure methods on “frozen” values.

How to use Java methods

- General syntax:
  `[pure] native v j :: t`
  - The idea is to define ***v*** by giving tiny code snippets ***j*** that can be completed by the code generator taking the type ***t*** into account.
  - if ***t*** is a function type, all arguments are assumed to be strict.
  - if ***v*** is the same as ***j***, one may omit ***j***
- Supports the following Java constructs:
  - constant, field access, method invocation, class instance creation, binary, unary and cast expressions.

Constants, static field access

- recognized by not having function types

```
native pi “3.14195” :: Double
```

```
final public static double pi = 3.14159;
```

  - Does not really make sense, it’s merely an unintended by-product. The quotes are needed because the java part must be a word or a (fully qualified) java identifier or a string.

```
pure native pi java.lang.Math.PI :: Double
```

```
final … double pi = java.lang.Math.PI;
```

  - fully qualified name, must be legal java expression.

Instance field access

```
data IntArr = native “int[]”
pure native länge “.length” :: IntArr -> Int
```

- rarely needed, as classes that expose instance fields are frowned upon.
- recognized by java code that starts with a dot and is followed by a simple name
- type must be of the form `ref -> a` where `ref` denotes a reference type. Everything that is not one of the primitive types is taken to be a reference type.

Method Invocation

- application of a native function will map to a method invocation expression
- recognized by having function type and not being one of the special forms like instance field access. The “receiver” is the first arg.

```
pure native sin java.lang.Math.sin
                         :: Double -> Double
pure native at charAt :: String -> Int -> Char
native read :: Reader -> IO Int
native write :: Writer -> String -> IO ()
```

Instance creation expression

- Has the java code "new" and a function type whose raw result is a reference type.

```
data Date = native java.util.Date where
    native new
        :: ()    -> IO (Mutable RealWorld Date)
        |  Long -> ST s (Mutable s Date)
    native getTime
        :: Mutable s Date -> ST s Long
data Integer = pure native java.math.BigInteger
    where pure native new :: String -> Integer
```

Magic Native Function Types

- `()` in argument position, must be only argument besides receiver, if any. Signals empty argument list.
- `()` as result signals `void` methods. Makes only sense when wrapped in `IO` or `ST`. If you need a pure function that always returns `()`, consider `const ()`
- `Maybe a` as argument allows to pass `null` instead of an actual value.
- `Maybe a` as result maps `null` results to `Nothing` and wraps non `null` results in `Just`. This is the way to deal with Java null values.
- `[a]` as result converts array or `Iterable` values to lists, skipping `null` elements. Also turns `null` into `[]`.

Rules for sound native function types

1. The only way native mutable type `M` can appear in the return type of any function is `Mutable s M`
2. A pure function may not return `Mutable s a`
3. An impure function must have return type `ST s a` or `IO a`.
4. The only way mutable type `M` can appear in the argument type of an impure function is `Mutable s M`
5. The phantom types after `ST` and `Mutable` must all be equal, either the same type variable, or `RealWorld`

This ensures that mutable values can only be created in the ST/IO Monad, from whence they cannot normally escape.

The rules need only be checked on native functions.

Tools

- fregec.jar contains command line compiler and standard library. JDK 7 required.
- FregIDE - eclipse plugin, contains its own copy of fregec.jar. Can be used with just a JRE 7.
- REPL and online REPL
- Documentation tool, creates HTML from Frege class files (included in fregec.jar).
- Quick check tool, checks quick check properties of class files (included in fregec.jar).

Links

- Project pages: github.com/Frege
  - The repositories for frege core, eclipse plugin, REPL etc. are linked there.
  - Wiki with Differences Frege/Haskell among other stuff
- Downloads: github.com/Frege/frege/releases
  - fregec.jar, FregIDE, Language reference.
- Discussions: on google groups
- Try Frege Online: try.frege-lang.org/
- Questions: on stackoverflow with tag “frege”
- Example code mentioned in slides, and more.