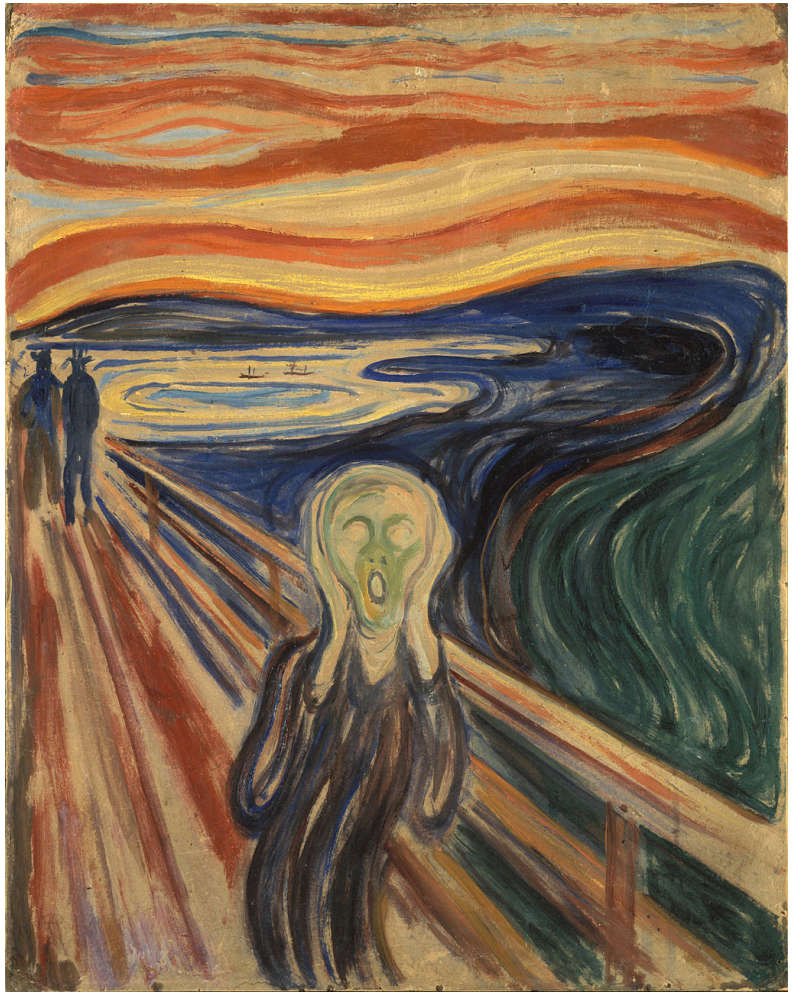
Why Functional Programming Matters[1]

Functional Programming Graz

April 2016

[1] Title stolen from (Hughes, 1984)

Functional programming is scary?



Functional programming is scary?

Don't fear: we *don't* mean code like this:[2]

```
gcata :: (Functor f, Comonad n) =>
  (forall a. f (n a) -> n (f a))
    -> (f (n c) -> c) -> Mu f -> c
gcata dist phi = extr . cata (fmap phi . dist . fmap dupl)

zygo chi = gcata (fork (fmap outl) (chi . fmap outr))

para :: Functor f => (f (Prod (Mu f) c) -> c) -> Mu f -> c
para = zygo In
```

[2] (Code by Fritz Ruehr)

No! (at least, mostly)

Rather, we would like to focus on stuff that makes code *more* readable:[3]

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort smaller
                   ++ [x]
                   ++ quicksort greater
  where smaller = [y | y <- xs, y <= x]
        greater = [y | y <- xs, y > x]
```

Of course, as with everything, you have to get used to it first. But. . .

[3] Note that in Haskell, function application works by juxtaposition, not parentheses; this is actually very useful with higher order functions.

Functional programming can lead to enlightening!

. . . or at least, it will change you way of thinking and approaching problems in a positive way.

Higher-order functions to the rescue!

If used in the right places, we can write more declarative and reusable code by using functions as values.

```
union = reduce(or_, all_sets, set())
hitting_sets = {p for p in powerset(union)
                if all(p & s for s in all_sets)}
```

Doing such things needs getting used to, and requires one to use much more recursion and less mutability than one normally would as an imperative programmer, but the resulting style has a lot of advantages.

Remember pipe & filter?

The usual example of stream processing: passing functions into `map`, `filter`, and friends:[4]

```
transactions.stream()
            .filter(t ->
              t.getType() == Transaction.GROCERY)
            .sorted(comparing(Transaction::getValue)
                    .reversed())
            .map(Transaction::getId)
            .collect(toList());
```

Although this quickly gets boring. . . but we can also do "non-linear" things.

[4] (Code by Raoul-Gabriel Urma)

Streams for more complex list processing

This expression does overload resolution for function arguments:

```
overloads.stream()
  .flatMap(o -> applicationCostFor(arguments, o.input)
                .map(c -> Stream.of(Pair.of(o.output, c)))
                .orElse(Stream.empty()))
  .sorted(Comparator.comparing(p -> p._2))
  .map(p -> p._1)
  .findFirst();
```

`flatMap` allows to filter and map at the same time; also, using `map` on `Optional`s (`applicationCostFor` returns an `Optional<Int>`) makes this one safe, self-contained expression.

Lazy streams

This gets much more interesting if we use non-strict evaluation:

```
Prelude> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
Prelude> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
```

Yes, that is a recursively defined list of infinite length.[5]

[5] Although this is inefficient; but there is an almost equivalent memoized version requiring only linear time and space.

Reactive streams

Taking this to the extreme, we arrive at reactive programming:[6]

```
var refreshClickStream =
  Rx.Observable.fromEvent(refreshButton, 'click');
var requestStream = refreshClickStream
  .startWith('startup click')
  .map(function() {
    var randomOffset = Math.floor(Math.random() * 500);
    return 'https://api.github.com/users?since='
      + randomOffset;
  });
  .flatMap(function (requestUrl) {
    return Rx.Observable.fromPromise(
      $.ajax({url: requestUrl}));
  });
```

[6] (Code by Andre Medeiros)

Function chaining in R (dplyr)

There is also a certain relationship with query languages; however, we can mix in arbitrary custom functions. This is a way to describe operations on data frames (in-memory, or possibly in a database):[7]

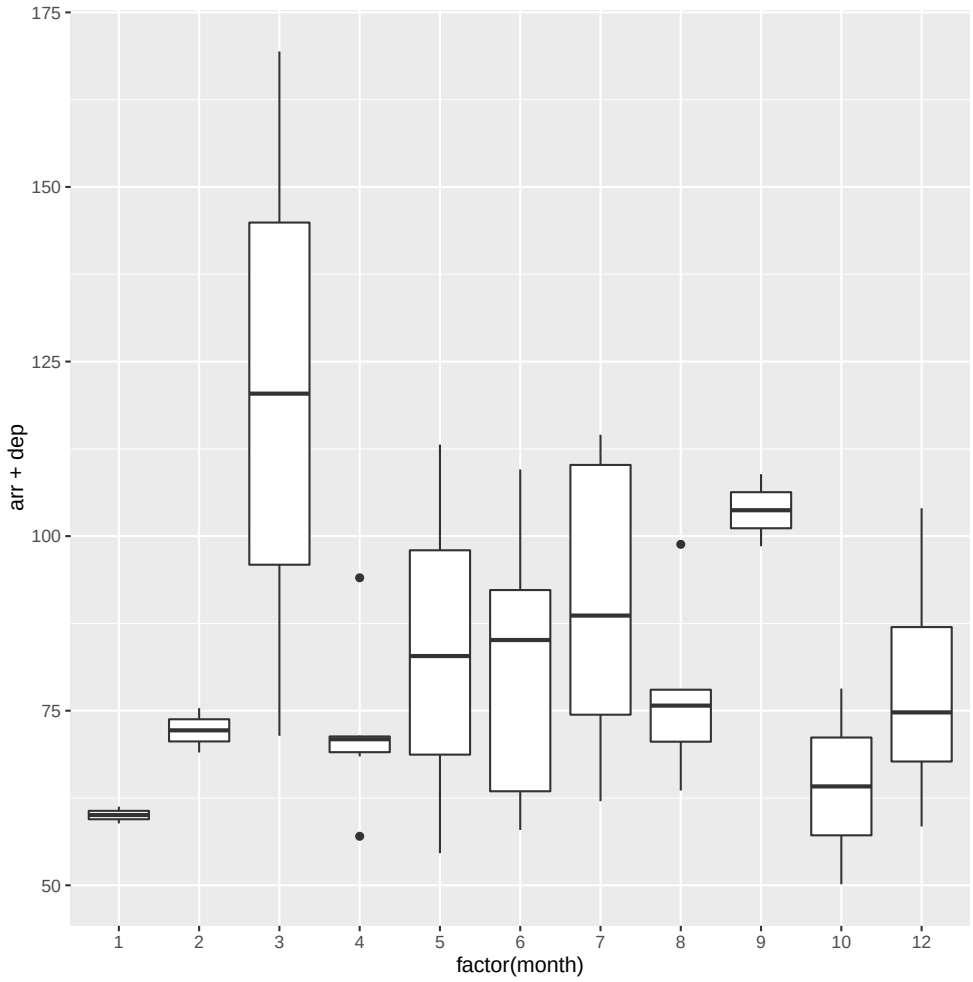
```
flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(arr = mean(arr_delay),
            dep = mean(dep_delay)) %>%
  filter(arr > 30 | dep > 30) %>%
  ggplot(aes(factor(month), arr + dep)) + geom_boxplot()
```

`x %>% f(y)` is turned into `f(x, y)`. We see that this pattern relies on immutability and partial application.

[7] (Code from the `dplyr` vignette)

Function chaining in R (`dplyr`)

Result:



Property-based testing

Testing by specification of invariants:

```
val propReverseList = forAll {
  l: List[String] => l.reverse.reverse == l
}

val propConcatString = forAll {
  (s1: String, s2: String) => (s1 + s2).endsWith(s2)
}
```

This is *not* a unit test – instead, the predicate is applied to a lot of randomly generated values of its input types.
How is `forAll` implemented? Purely as a library function![8]

[8] See: `http://www.scalacheck.org/`

Algebraic data types (ADTs)

...are lightweight and extremely useful. Especially, we have *sum types* (aka "tagged unions"). They are used through pattern matching:

```
data Bool = True | False

(||) :: Bool -> Bool -> Bool
True || something = True
False || something = False

data Tree a = Leave a | Branch (Tree a) (Tree a)

contains :: (a -> Bool) -> Tree a -> Bool
contains p (Leave a) = p a
contains p (Branch l r) = contains p l || contains p r
```

(As visible in the type, `p` is a predicate, ie. a function from `a` to `Bool`.)

Option: the better `null`

Now, what if we also want to get the matched value, not only find out whether it's there? We must be careful, since maybe there is no value at all. To represent that option, we use `Option`:

```
data Option a = Nothing | Just a

find :: (a -> Bool) -> Tree a -> Option a
find p (Leave a) = if p a then Just a
                          else Nothing
find p (Branch l r) = case (find p l) of
  Just x -> Just x
  Nothing -> find p r
```

Option: the better `null`

But this nesting quickly gets tedious. Therefore: combinators, and more syntax.

```
find' :: (a -> Bool) -> Tree a -> Option a
find' p (Leave a) | (p a) = Just a
                  | otherwise = Nothing
find' p (Branch l r) = (find p l) <|> (find p r)
```

The (custom) operator `<|>` takes the left value, if it is not `Nothing`, and otherwise returns the right side:

```
(<|>) :: Option a -> Option a -> Option a
(Just x) <|> something = Just x
Nothing <|> something = something
```

ADTs in object orientation

Scala nicely integrates algebraic data types into its object oriented type system:

```
sealed trait Expr
case class Var(name: String) extends Expr
case class App(l: Expr, r: Expr) extends Expr
case class Lambda(param: String, body: Expr) extends Expr

def freeVarsOf(e: Expr): Set[String] = e match {
  case Var(x) => Set(x)
  case App(t1, t2) => freeVarsOf(t1) ++ freeVarsOf(t2)
  case Lambda(x, t) => freeVarsOf(t) - x
}
```

Let the compiler do the work for you

It's not like non-functional languages wouldn't have powerful type systems, too:[9]

```
template<typename T>
T adder(T v) {
  return v;
}

template<typename T, typename... Args>
T adder(T first, Args... args) {
  return first + adder(args...);
}
```

But note that the above code is extremely funtional. In fact, the research about type systems is closely related to functional programming, since these two concepts naturally play well together – whereas in many imperative languages, sophisticated types are more like a construct put upon them after the fact.

[9] (Code by Ellen Bendersky)

Usage of type systems

For one, they enable very abstract reuse (by sometimes "extreme" polymorphism):[10]

```
-- | `until p f` yields the result of applying f
-- until p holds.
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if (p x) then x
              else until p f (f x)

-- | The sum of a collection of actions,
-- generalizing 'concat'.
asum :: (Foldable t, Alternative f) => t (f a) -> f a
asum = foldr (<|>) empty
```

[10] (From the Haskell prelude)

Usage of type systems

On the other hand, there are interesting ways to achieve compile-time safety:

```
sealed trait Empty
sealed trait NonEmpty

sealed trait SafeList[Tag, +A]
case object SafeNil extends SafeList[Empty, Nothing]
case class SafeCons[+A](head: A, tail: SafeList[_, A])
  extends SafeList[NonEmpty, A]

def safeHead[A](xs: SafeList[NonEmpty, A]): A = xs match {
  case SafeCons(a, something) => a
}
```

Testing:

```
scala> safeHead(SafeCons(42, SafeNil))
res2: Int = 42
scala> safeHead(SafeNil)
<console>:18: error: type mismatch;
  found   : SafeNil.type
  required: SafeList[NonEmpty,?]
```

Let the compiler do the work for you

Often, when we know what type a function must have, we immediately know how it is implemented:

```
mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Leave a) = Leave (f a)
mapTree f (Branch l r) = Branch (mapTree f l) (mapTree f r)
```

For the given type signature, and the definition of `Tree`, there exists only one possible implementation of this function, which can even be derived automatically.[11]

[11] (see Wadler, 1989)

Type-driven development

Let's say we want to develop a parser. Now, what *is* a parser, actually? We can treat is as a function taking a string and returning something that is parsed, together with the (yet unparsed) rest of the input. And since parsing can fail, we should wrap the result, say, in `Option`:

```
case class Parser[A](run: String => Option[(A, String)])
```

Type-driven development

The simplest thing we need: parsing a single character. The function doing this should be obvious:

```
def char(c: Char): Parser[Char] = Parser {
  case "" => None
  case input => if (input.head == c) Some(c, input.tail)
                else None
}

scala> char('x').run("xyz")
res11: Option[(Char, String)] = Some((x,yz))
scala> char('x').run("yzx")
res12: Option[(Char, String)] = None
```

Type-driven development

The next important thing is sequencing, alternation, and repetition. We can derive these almost automatically from the types they need to have.
Alternation:

```
def or[A](p1: Parser[A], p2: Parser[A]): Parser[A] = Parser
  input => p1.run(input) match {
    case None => p2.run(input)
    case something => something
  }
}

scala> or(char('a'), char('b')).run("bcd")
res14: Option[(Char, String)] = Some((b,cd))
```

Type-driven development

Sequencing:

```
def chain[A, B](p1: Parser[A],
                p2: Parser[B]): Parser[(A, B)] = Parser {
  input => p1.run(input) match {
    case None => None
    case Some((a, rest)) => p2.run(rest) match {
      case None => None
      case Some((b, rest2)) => Some((a, b), rest2)
    }
  }
}

scala> chain(char('a'), char('b')).run("abc")
res19: Option[((Char, Char), String)] = Some(((a,b),c))
```

Type-driven development

Repetition:

```
def many[A](p: Parser[A]): Parser[List[A]] = Parser {
  input => p.run(input) match {
    case None => Some(List(), input)
    case Some((a, rest)) => many(p).run(rest) match {
      case None => Some(List(a), rest)
      case Some((tail, rest)) => Some(a::tail, rest)
    }
  }
}

scala> many(or(char('a'), char('b'))).run("abbbabcaab")
res18: Option[(List[Char], String)]
  = Some((List(a, b, b, b, a, b),caab))
```

Type-driven development

Now we have regular expressions:

```
scala> val p = chain(chain(char('a'),
     |                     many(or(char('b'), char('c')))),
     |               char('d'))

scala> p.run("ad")
res25: Option[(((Char, List[Char]), Char), String)]
  = Some((((a,List()),d),))

scala> p.run("abcd")
res26: Option[(((Char, List[Char]), Char), String)]
  = Some((((a,List(b, c)),d),))

scala> p.run("abbcbbcccbcdefg")
res27: Option[(((Char, List[Char]), Char), String)]
  = Some((((a,List(b, b, c, b, b, c, c, c, b, c)),d),efg))
```