Monadologie —

professional help for type anxiety

Christopher League

12 July 2010

Monadologie —

github.com/league/scala-type-anxiety

@chrisleague

league@contrapunctus.net

1996


GJ
GJ

Pizza into Java: Translating theory into practice

Martin Odersky
University of Karlsruhe

Philip Wadler
University of Glasgow

Abstract

Pizza is a strict superset of Java that incorporates three ideas from the academic community: parametric polymorphism, higher-order functions, and algebraic data types. Pizza is defined by translation into Java and compiles into the Java Virtual Machine, requirements which strongly constrain the design space. Nonetheless, Pizza fits smoothly to Java, with only a few rough edges.

1 Introduction

There is nothing new beneath the sun.
— *Ecclesiastes 1:10*

Java embodies several great ideas, including:

- strong static typing,
- heap allocation with garbage collection, and
- safe execution that never corrupts the store.

These eliminate some sources of programming errors and enhance the portability of software across a network.

- algebraic data types.

Pizza attempts to make these ideas accessible by translating them into Java. We mean that both figuratively and literally, because Pizza is defined by translation into Java. Our requirement that Pizza translate into Java strongly constrained the design space. Despite this, it turns out that our new features integrate well: Pizza fits smoothly to Java, with relatively few rough edges.

Promoting innovation by extending a popular existing language, and defining the new language features by translation into the old, are also not new ideas. They have proved spectacularly successful in the case of C++.

Pizza and Java. Our initial goal was that Pizza should compile into the Java Virtual Machine, or JVM. We considered this essential because the JVM will be available across a wide variety of platforms, including web browsers and special-purpose chips. In addition, we required that existing code compiled from Java should smoothly inter-operate with new code compiled from Pizza. Among other things, this would give Pizza programmers access to the extensive Java libraries that exist for graphics and networking.

We did not originally insist that Pizza should translate into Java, or that Pizza should be a superset of Java. However, it soon became apparent that the JVM and Java were

Pizza into Java:
Translating theory into practice

Martin Odersky
University of Karlsruhe

Philip Wadler
University of Glasgow

Abstract

Pizza is a strict superset of Java that incorporates three ideas from the academic community: parametric polymorphism, higher-order functions, and algebraic data types. Pizza is defined by translation into Java and compiles into the Java Virtual Machine, requirements which strongly constrain the design space. Nonetheless, Pizza fits smoothly to Java, with only a few rough edges.

1 Introduction

There is nothing new beneath the sun.
— *Ecclesiastes 1:10*

Java embodies several great ideas, including:

- strong static typing,
- heap allocation with garbage collection, and
- safe execution that never corrupts the store.

These eliminate some sources of programming errors and enhance the portability of software across a network.

Principles of Programming Languages

January 1997 § Paris

Pizza into Java:
Translating theory into practice

Martin Odersky
University of Karlsruhe

Philip Wadler
University of Glasgow

"Java might be the vehicle that will help us carry modern programming language innovations into industrial practice."

— *me*

Principles of Programming Languages

January 1997 § Paris

Scala

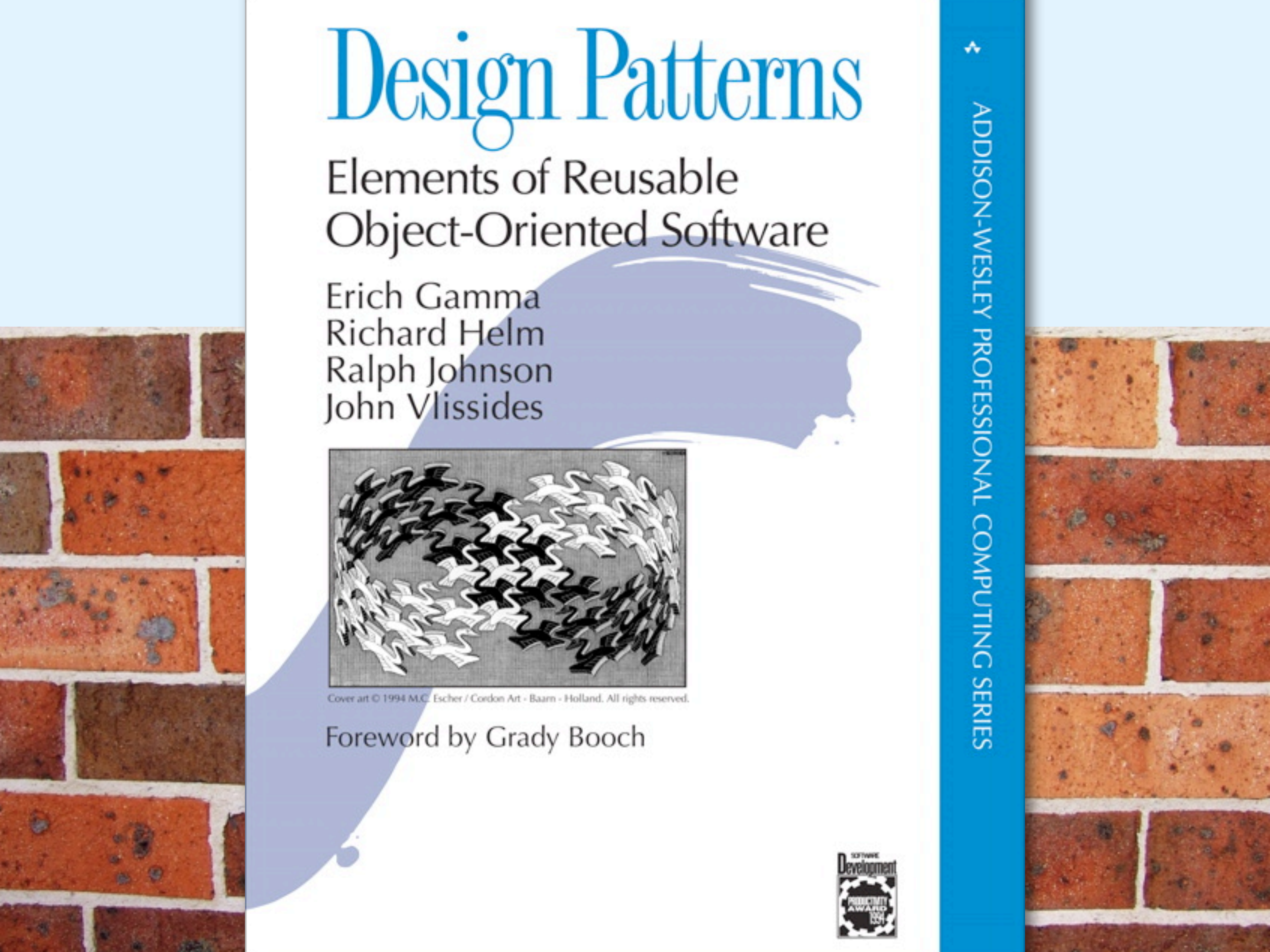
Monadologie

continuations
monads
existentials
variance
effects

Monadologie

continuations — control flow
monads — data flow

$$\vdash \tau$$


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Continuations

Continuation-Passing Style

```scala
def plus [A] (x: Int, y: Int, k: Int=>A): A =
    k(x+y)

def times [A] (x: Int, y: Int, k: Int=>A): A =
    k(x*y)

def less [A] (x: Int, y: Int,
              kt: => A, kf: => A): A =
    if(x < y) kt else kf
```

```scala
def factorial [A] (n: Int, k: Int => A): A =
    less(n, 2, k(1),
         plus(n, -1, (m:Int) =>
              factorial(m, (f:Int) =>
                        times(n, f, k))))
```

```
scala> factorial(5, println)
120
scala> factorial(3, factorial(_, println))
720
scala> val n = factorial(3, r => r)
n: Int = 6
```

CPS

makes some programs simpler

```scala
class Regex
case class Literal(s: String)         extends Regex
case class Concat(r1: Regex, r2: Regex)
                                      extends Regex
case class Choice(r1: Regex, r2: Regex)
                                      extends Regex
case class Star(r: Regex)             extends Regex
```

```scala
Concat(Star(Literal("ab")), Literal("abc"))
// (ab*)abc  matches  abababc

Choice(Star(Literal("ab")),
       Concat(Literal("a"), Star(Literal("ba"))))
// (ab)*|a(ba)*  matches  abababa
```

```scala
def accept (regex: Regex, chars: Seq[Char],
            k: Seq[Char] => Boolean): Boolean =
  regex match {
    case Literal(expect) =>
      if(chars.take(expect.length).sameElements(expect))
        k(chars.drop(expect.length))
      else false
    case Concat(regex1, regex2) =>
      accept(regex1, chars, remaining =>
        accept(regex2, remaining, k))
    case Choice(regex1, regex2) =>
      accept(regex1, chars, k) || accept(regex2, chars, k)
    case Star(repeatable) =>
      k(chars) ||
      accept(repeatable, chars, remaining =>
        accept(regex, remaining, k))
  }
```

```scala
def accept (regex: Regex, chars: Seq[Char],
            k: Seq[Char] => Boolean): Boolean =
  ...

def complete (remaining: Seq[Char]): Boolean =
    remaining.length == 0

accept(regex1, "abababc", complete) // true
accept(regex2, "abababa", complete) // true
accept(regex1, "abababcd", complete) // false
```

Delimited Continuations

via compiler plugin (2.8)

Delimited Continuations

shift & reset

```scala
def doSomething0 = reset {
  println("Ready?")
  val result = 1 +            * 3
  println(result)
}
```

Punch a hole *in your code.*

What type of value is expected in the hole?

What happens after *the hole?*

What is result type of the reset *block?*

```
def doSomething0 = reset {
  println("Ready?")
  val result = 1 + [        ] * 3
  println(result)
}
```

Think of the rest *of the computation as a* function *with the hole as its parameter. Call it the* continuation.

```scala
def doSomething1 = reset {
  println("Ready?")
  val result = 1 + special * 3
  println(result)
}

def special = shift {
  k: (Int => Unit) =>
    println(99)
    "Gotcha!"
}
```

shift *captures the continuation and then determines its* own *future.*

```scala
def doSomething1 = reset {
  println("Ready?")
  val result = 1 + special * 3
  println(result)
}

def special = shift {
  k: (Int => Unit) =>
    println(99)
    "Gotcha!"
}
```

```
scala> doSomething1
Ready?
99
res0: java.lang.String = Gotcha!
```

```scala
def doSomething2 = reset {
  println("Ready?")
  val result = 1 + wacky * 3
  println(result)
}

def wacky = shift {
  k: (Int => Unit) =>
    k(2)
    println("Yo!")
    k(3)
}
```

```scala
def doSomething2 = reset {
  println("Ready?")
  val result = 1 + wacky * 3
  println(result)
}

def wacky = shift {
  k: (Int => Unit) =>
    k(2)
    println("Yo!")
    k(3)
}
```

```
scala> doSomething2
Ready?
7
Yo!
10
```

Continuation

low-level control-flow primitive that can implement:

exceptions
concurrency (actors)
suspensions (lazy)
...

```
def multi = reset {
  println("This function returns tentatively")
  println("but you can always ask for more.")
  produce(42)
  println("Didn't like that? Back again.")
  produce(99)
  println("Still not OK? I'm out of ideas!")
  None
}
```

```scala
def multi = reset {
  println("This function returns tentatively")
  println("but you can always ask for more.")
  produce(42)
  println
  produce
  println
  None
}
```

```
scala> multi
This function returns tentatively
but you can always ask for more.
res0: Option[ReturnThunk[Int]] = Some(...)
scala> println(res0.get.value)
42
scala> res0.get.proceed
Didn't like that? Back again.
res1: Option[ReturnThunk[Int]] = Some(...)
scala> println(res1.get.value)
99
scala> res1.get.proceed
Still not OK? I'm out of ideas!
res2: Option[ReturnThunk[Int]] = None
```

```
def multi = reset {
  println("This function returns tentatively")
  println("but you can always ask for more.")
  produce(42)
  println("Didn't like that? Back again.")
  produce(99)
  println("Still not OK? I'm out of ideas!")
  None
}
```

```scala
case class ReturnThunk[A]
  (value: A,
   proceed: Unit => Option[ReturnThunk[A]])

def produce [A] (value: A):
    Unit @cps[Option[ReturnThunk[A]]] =
  shift {
    k: (Unit => Option[ReturnThunk[A]]) =>
      Some(ReturnThunk(value, k))
  }
```

```scala
def multi = reset {
  println("This function returns tentatively")
  println("but you can always ask for more.")
  produce(42)
  println("Didn't like that? Back again.")
  produce(99)
  println("Still not OK? I'm out of ideas!")
  None
}
```

```scala
def interact = reset {
  val first = ask("Please give me a number")
  val second = ask("Enter another number")
  printf("The sum of your numbers is: %d\n",
         first + second)
}
```

```scala
def interact = reset {
  val first = ask("Please give me a number")
  val second = ask("Enter another number")
  printf("The sum of your numbers is: %d\n",
         first + second)
}
```

```
scala> interact
Please give me a number
respond with: submit(0x28d092b7, ...)
scala> submit(0x28d092b7, 14)
Enter another number
respond with: submit(0x1ddb017b, ...)
scala> submit(0x1ddb017b, 28)
The sum of your numbers is: 42
```

```scala
type UUID = Int
def uuidGen: UUID = Random.nextInt
type Data = Int
val sessions = new HashMap[UUID, Data=>Unit]

def ask(prompt: String): Data @cps[Unit] = shift {
    k: (Data => Unit) => {
      val id = uuidGen
      printf("%s\nrespond with: submit(0x%x, ...)\n",
             prompt, id)
      sessions += id -> k
    }
}

def submit(id: UUID, data: Data) = sessions(id)(data)
```

```scala
def interact = reset {
  val first = ask("Please give me a number")
  val second = ask("Enter another number")
  printf("The sum of your numbers is: %d\n",
         first + second)
}
```

WARNING

HARMFUL

```scala
def harmful = reset {
    var i = 0
    println("Hello world!")
    label("loop")
    println(i)
    i = i + 1
    if(i < 20) goto("loop")
    println("Done.")
}
```

```scala
def harmful = reset {
    var i = 0
    println("Hello world!")
    label("loop")
    println(i)
    i = i + 1
    if(i < 20) goto("loop")
    println("Done.")
}
```

```
Hello world!
0
1
2
.
.
.
18
19
Done.
```

```scala
def harmful = reset {
    var i = 0
    println("Hello world!")
    label("loop")
    println(i)
    i = i + 1
    if(i < 20) goto("loop")
    println("Done.")
}
```

```scala
val labelMap = new HashMap[String, Unit=>Unit]

def label(name:String) =
  shift { k:(Unit=>Unit) =>
    labelMap += name -> k
    k()
  }
def goto(name:String) =
  shift { k:(Unit=>Unit) => labelMap(name)() }
```

```scala
def harmful = reset {
    var i = 0
    println("Hello world!")
    label("loop")
    println(i)
    i = i + 1
    if(i < 20) goto("loop")
    println("Done.")
}
```

Monads

the leading design pattern
for functional programming

A type constructor M *is a* monad
if it supports these operations:

```
def unit[A] (x: A): M[A]

def flatMap[A,B] (m: M[A]) (f: A => M[B]): M[B]

def map[A,B] (m: M[A]) (f: A => B): M[B] =
  flatMap(m){ x => unit(f(x)) }

def andThen[A,B] (ma: M[A]) (mb: M[B]): M[B] =
  flatMap(ma){ x => mb }
```

Option *is a monad.*

```
def unit[A] (x: A): Option[A] = Some(x)

def flatMap[A,B](m:Option[A])(f:A =>Option[B]):
      Option[B] =
    m match {
      case None => None
      case Some(x) => f(x)
    }
```

List *is a monad.*

```scala
def unit[A] (x: A): List[A] = List(x)

def flatMap[A,B](m:List[A])(f:A =>List[B]): List[B] =
  m match {
    case Nil => Nil
    case x::xs => f(x) ::: flatMap(xs)(f)
  }
```

For comprehension: convenient syntax for monadic structures.

```
for(i <- 1 to 4;
    j <- 1 to i;
    k <- 1 to j)
yield { i*j*k }
```

Compiler translates it to:

```
(1 to 4).flatMap { i =>
  (1 to i).flatMap { j =>
    (1 to j).map { k =>
      i*j*k }}}
```

Example: a series of operations, where each may fail.

```
lookupVenue: String => Option[Venue]
getLoggedInUser: SessionID => Option[User]
reserveTable: (Venue, User) => Option[ConfNo]
```

Example: a series of operations, where each may fail.

```
lookupVenue: String => Option[Venue]
getLoggedInUser: SessionID => Option[User]
reserveTable: (Venue, User) => Option[ConfNo]

val user = getLoggedInUser(session)
val confirm =
  if(!user.isDefined) { None }
  else lookupVenue(name) match {
    case None => None
    case Some(venue) => {
      val confno = reserveTable(venue, user.get)
      if(confno.isDefined)
        mailTo(confno.get, user.get)
      confno
    }
  }
```

Example: a series of operations, where each may fail.

```
lookupVenue: String => Option[Venue]
getLoggedInUser: SessionID => Option[User]
reserveTable: (Venue, User) => Option[ConfNo]

val user = getLoggedInUser(session)
val confirm =
  if(!user.isDefined) { None }
  else lookupVenue(name) match {
    case None => None
    case Some(venue) => {
      val confno = reserveTable(venue, user.get)
      if(confno.isDefined)
        mailTo(confno.get, user.get)
      confno
    }
  }
```

Example: a series of operations, where each may fail.

```
lookupVenue: String => Option[Venue]
getLoggedInUser: SessionID => Option[User]
reserveTable: (Venue, User) => Option[ConfNo]

val confirm =
  for(venue <- lookupVenue(name);
      user <- getLoggedInUser(session);
      confno <- reserveTable(venue, user))
  yield {
    mailTo(confno, user)
    confno
  }
```

Example from Lift *form validation:*

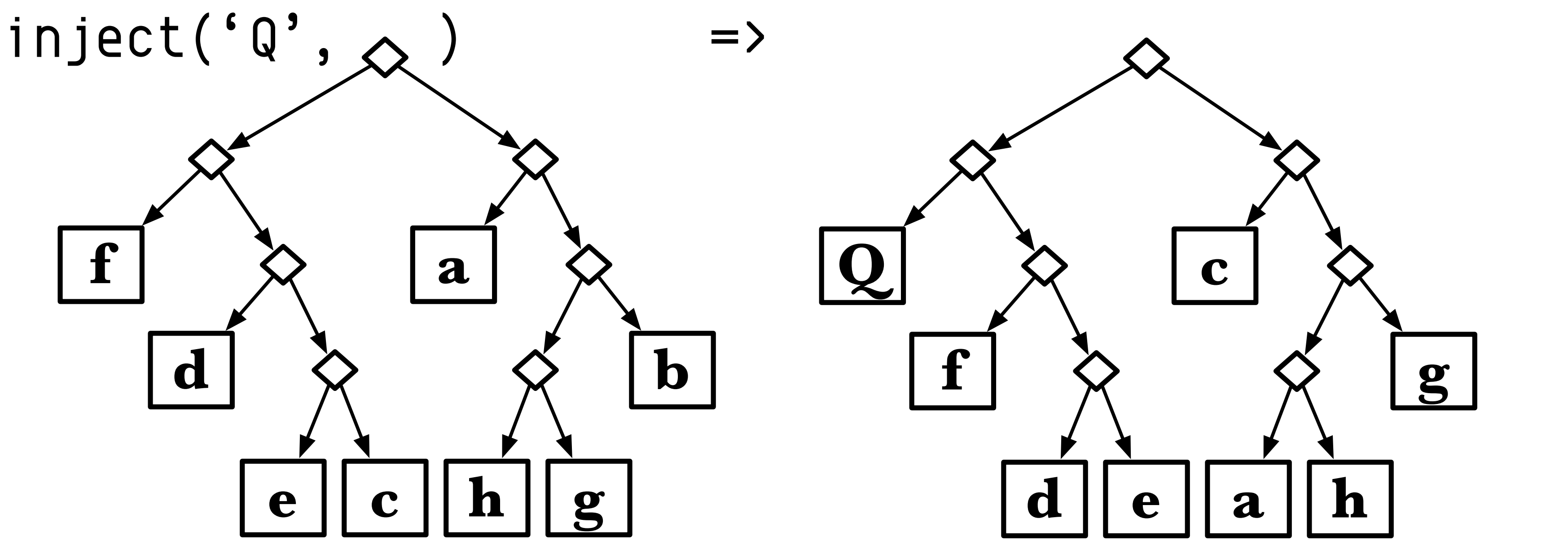
```
def addUser(): Box[UserInfo] =
  for {
    firstname <- S.param("firstname") ?~
                 "firstname parameter missing" ~> 400
    lastname <- S.param("lastname") ?~
                 "lastname parameter missing"
    email <- S.param("email") ?~
                 "email parameter missing"
  } yield {
    val u = User.create.firstName(firstname).
    lastName(lastname).email(email)

    S.param("password") foreach u.password.set

    u.saveMe
  }
```

Example: use monad to pass around state behind the scenes

```scala
class Tree[A]
case class Leaf[A](elem: A) extends Tree[A]
case class Branch[A](left: Tree[A], right: Tree[A])
        extends Tree[A]
```



```scala
def inject[A] (root: Tree[A], cur: A): (Tree[A], A) =
    root match {
      case Leaf(old) => (Leaf(cur), old)
      case Branch(left, right) =>
        val (t1, last1) = inject(left, cur)
        val (t2, last2) = inject(right, last1)
        (Branch(t1,t2), last2)
    }
```

```scala
def inject[A] (root: Tree[A], cur: A): (Tree[A], A) =
    root match {
      case Leaf(old) => (Leaf(cur), old)
      case Branch(left, right) =>
        val (t1, last1) = inject(left, cur)
        val (t2, last2) = inject(right, last1)
        (Branch(t1,t2), last2)
    }
```

```scala
def injectST[A] (root: Tree[A]): ST[A, Tree[A]] =
    root match {
      case Leaf(old) =>
        for(cur <- init[A];
            u <- update[A](_ => old))
          yield Leaf(cur)
      case Branch(left, right) =>
        for(t1 <- injectST(left);
            t2 <- injectST(right))
          yield Branch(t1,t2)
    }
```

```scala
case class ST[S,A](exec: S => (S,A)) {
  def flatMap[B] (f: A => ST[S,B]): ST[S,B] =
    ST { s0 =>
      val (s1, a) = exec(s0)
      f(a).exec(s1)
      }

  def map[B] (f: A => B)
        (implicit unit: B => ST[S,B]) : ST[S,B] =
    flatMap { x => unit(f(x)) }
}

implicit def unitST[S,A] (x: A): ST[S,A] =
  ST { s0 => (s0, x) }

def init[S]: ST[S,S] =
  ST { s0 => (s0, s0) }

def update[S] (g: S => S): ST[S,Unit] =
  ST { s0 => (g(s0), ()) }
```

```scala
case class ST[S,A](exec: S => (S,A)) {
  def flatMap[B] (f:
    ST { s0 =>
      val (s1, a) = (
      f(a).exec(s1)
      }

  def map[B] (f: A =>
        (implicit un
    flatMap { x => u
}

implicit def unitST[
  ST { s0 => (s0, x)

def init[S]: ST[S,S]
  ST { s0 => (s0, s0

def update[S] (g: S =
  ST { s0 => (g(s0),
```

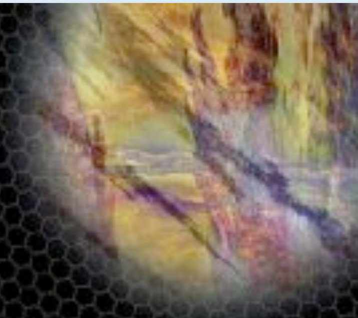
```
scala> drawTree(t1,"")
______f
|  \_____d
|     \_____e
|        \__c
\_____a
   \________h
      |  \__g
      \__b

scala> val m = injectST(t1)
m: ST[Char,Tree[Char]] = ST(<function1>)

scala> val (_,t2) = m.exec('Q')
t2: Tree[Char] = ...

scala> drawTree(t2,"")
______Q
|  \_____f
|     \_____d
|        \__e
\_____c
   \________a
      |  \__h
      \__g
```

THE EXPERT'S VOICE®



Pro Scala

Monadic Design Patterns for the Web

Gregory Meredith

Apress®

Monadologie

github.com/league/scala-type-anxiety

@chrisleague

league@contrapunctus.net