How do functional programmers do dependency injection?

Mark Hopkins

ScalaSyd 8 June 2006

Program to the interface, not the implementation

Abstraction helps us write cleaner, more modular code.

More flexible

We can swap out one implementation for another (e.g. for testing)
i.e. polymorphism

How to app, imperative style

- write app (using a set of interfaces)
- make a choice for each implementation we require
- “wire” these into our components from outside
  (because forcing each component know about the global config would be ugly and inflexible)
  i.e. **inject dependencies**
- run

In the Scala ecosystem, some people use existing Java DI frameworks

e.g. Spring, Guice.

Others have written new ones taking advantage of Scala language features:

e.g. Macwire, Scaldi, SubCut.

There's also a school of thought that realises if we're doing FP then objects really aren't that relevant. . .

Why bother wiring them up at all?

Prefer values, and functions (that produce values).

And there's one really obvious way of giving values to a function: pass them as arguments.

```scala
class MyClass(d1: Dep1, d2: Dep2, d3: Dep3) {
    def getResult: R = ...

    override def toString = ...
    override def equals   = ...
    override def hashCode = ...
}

val result = new MyClass(myDep1, myDep2, myDep3).getResult
```

```
def result(d1: Dep1, d2: Dep2, d3: Dep3): R = ...
```

Reader monad

Hide the noise of passing all those parameters around.

This business of passing objects around so we can call methods on them is still *a little too concrete*.

Let's back up a little and see what we were trying to achieve.

We wanted to write programs with abstract operations, so that we weren't overly tied to a particular provider of those operations and its implementation details.
Let's rephrase this slightly and say we want to

- write our program using a language (DSL, "instruction set") of operations
- later on, interpret it using a (choice of) interpreter for our DSL

This notion encompasses, but is more general than dependency injection.

For example, instead of running my program, I might

- pretty-print it
- estimate its runtime
- calculate some statistics based on its structure
- serialise it
- save it to a database
- send it to another computer to execute
- rearrange and optimise it prior to execution
- step through, instruction by instruction, pausing and resuming
- compile it to another language (e.g. C)

Implementations also aren't limited to calling a method on an object.

What else do we want?

Modularity

We're going to want modular languages.

e.g. a DB language, a logging language, a Twitter language, . . .

Not all squashed together.

More subtly, we want to be able to extend a language by adding new operations.

And implement a language in terms of a smaller core language.

Another way to think of this: we want to be able to “add” languages (and interpreters).

Stable client code

If I write a program, then extend the language, the program should still be valid in the new language.

Modular interpreters

We don't want to have to rewrite an interpreter to be able to use it with others.

This is (a strong version of) the famous **expression problem**.

We're doing FP. . .

. . . so we'd better make sure this will play well with with monads

. . . and comonads

. . . and arrows.

Monads in a minute (and comonads and arrows)

They are all design patterns for talking about sequencing, binding and effects.
They arise naturally in a functional programming setting.

The functional programming vision

Better software through programs we can reason about.
No matter how complex our programs their behaviour remains predictable.

Referential transparency

Functions are values: their meaning does not depend on their context.

Equational reasoning for functions

Function composition obeys simple rules that allow us to reason about our code and refactor safely.

```
f compose identity = f = identity compose f
```

```
f compose (g compose h) = (f compose g) compose h
```

But this is all pure functions and values.
No disk, network, console, errors, missile launches.

How can we add side effects but retain predictability?

Moggi's great insight[1] was to use the type system.

`M[A]` will represent a computation that produces a value of type `A` but modulated by some "effect" described by `M`.

[1]Eugenio Moggi, Notions of computation as monads, 1991

Looking at our function signature
`f: A => B`
there are three things we can modify:

▶ source	`A`
▶ target	`B`
▶ arrow	`=>`

Each different thing we can wrap gives us one of the three basic type classes for structuring computations:

▶ comonads	`W[A] =>  B`
▶ monads	`  A  =>  M[B]`
▶ categories (and arrows)	`  A  >>> B`

Equational reasoning

Reinstating the rules that we had for functions gives us the **laws** these have to obey:

```
f =>= extract  = f =  extract =>= f
f >=> pure     = f =  pure    >=> f
f >>> identity = f = identity >>> f

(f =>= g) =>= h = f =>= (g =>= h)
(f >=> g) >=> h = f >=> (g >=> h)
(f >>> g) >>> h = f >>> (g >>> h)
```

This generalises beyond side effects.
We use **effect** (or **context**) as a way of talking about whatever "enhancement" `M[A]` brings over the raw type `A`.
e.g.

- `List[A]` adds "multiplicity" to `A`
- `Option[A]` adds "partiality" to `A`
- `Future[A]` adds "postponement of results" to `A`

We want to reuse these patterns that FP has evolved to deal with effects, sequencing and binding.

Effects in the specification

Most literature talks about a pure specification with effectful interpreters.
But sometimes, creating or handling effects belongs in the specification.
e.g. we want to

- fail early if some condition is not met.
- clean up after some operation

And this needs to obey appropriate laws.

Two basic approaches

- Make DSL and program into a data structure
- Keep DSL as a trait

Solution 1: Data structures

Datatypes a la carte, Wouter Swierstra, 2008

Motto:

> *Turn operations into constructors*

Let's start with a simple typed language of console interactions. You can

- *ask* the user something (and get a response)
- *tell* them something (and not expect a response).

Following our motto, let's express it as a data type.

```
sealed trait Console[A]
case class Ask(s: String) extends Console[String]
case class Tell(s: String) extends Console[Unit]
```

Each operation becomes a constructor.

Yoneda embedding

CPS transform to get something we can write a Functor instance for. Each constructor gets a continuation parameter specifying **what to do next**.

```
sealed trait Console[A]
case class Ask[A](s: String, next: String => A)
    extends Console[A]
case class Tell[A](s: String, next: Unit => A)
    extends Console[A]

implicit val functor = new Functor[Console] {
  def map[A, B](c: Console[A])(f: A => B): Console[B] =
    c match {
      case Ask(s, next)  => Ask(s, next andThen f)
      case Tell(s, next) => Tell(s, next andThen f)
    }
  }
```

This step can actually be automated (see `Coyoneda`).

Adding functors

So far, so good. What about modularity?
Idea: combining languages can literally be a **sum** of functors.

```
sealed trait Coproduct[F[_], G[_], A]

case class Inl[F[_], G[_], A](fa: F[A])
  extends Coproduct[F,G,A]
case class Inr[F[_], G[_], A](ga: G[A])
  extends Coproduct[F,G,A]
```

Morally, this is `F + G`, but Scala won't let us get away with anything so simple and natural.

A sum of two functors is naturally a functor. . .

```
implicit def functor[F[_]:Functor, G[_]:Functor] =
 new Functor[Coproduct[F, G, ?]] {
   def map[A, B]
    (c: Coproduct[F, G, A])(f: A => B): Coproduct[F, G, B]
     c match {
       case Inl(fa) => Inl(fa map f)
       case Inr(ga) => Inr(ga map f)
     }
 }
```

Now we can break `Console` up

```
case class Ask[A](s: String, next: String => A)
case class Tell[A](s: String, next: Unit => A)

implicit val askFunctor = ...
implicit val tellFunctor = ...
```

and put it back together again as a sum of its parts

```
type Console[A] = Coproduct[Ask, Tell, A]
```

Free monads

We can make a monad out of any functor, using the “free monad” construction.

- What is it, and where does it come from?
- What does “free” mean?
- Why is this important for interpreting monadic DSLs?

It turns out that being “free” is closely connected to our idea of operations as data types.

Let's see how to build `Free`.

Can we use “operations into constructors” to turn the Monad class into a data type?
Yes!

```
trait Monad[M[_]] {
  def pure[A](a: A): M[A]
  def join[A](mma: M[M[A]]): M[A]
}
```

becomes

```
sealed trait Free[F[_],A]

case class Pure[F[_],A](a: A) extends Free[F,A]
case class Join[F[_],A](ffa:F[Free[F,A]]) extends Free[F,A
```

`Free[F,A]` is a monad whenever `F` is a functor.
We said before this was the “free monad on `F`”.
What does this mean and why do we care?

Small digression into abstract nonsense.

Categories have **objects**, and **morphisms** between objects.

Let’s have a look at some categories

Scala types

Scala types form a category: a morphism between two types is just a function.

```
f: A => B
```

Monoids

Monoids form a category.
A monoid is just a set with an (associative) operation defined on it, together with a neutral element.
e.g.

- strings and concatenation
- numbers and $+$
- numbers and *
- matrices and $+$
- matrices and *
- polynomials over a monoid

A morphism of monoids is a function that preserves the operation.
e.g. `length` is a monoid morphism from `String` to `Int`, because

```
length(s + t) = length(s) + length(t)
```

Vector spaces

Vector spaces and linear functions form a category.
A linear function from V to W is a function obeying

$$f(av_1 + bv_2) = a(fv_1) + b(fv_2).$$

Functors

Functors between Scala types forms a category. A morphism between two functors is a **natural transformation**.
A natural transformation is just a polymorphic function

```
trait ~>[F[_], G[_]] {
  def apply[A](fa: F[A]): G[A]
```

Monads

Monads on scala types also form a category.
A monad is a functor with some additional structure (`flatMap` and `pure`).

So a morphism between monads will be a natural transformation preserving that structure:

```
n compose (f >=> g) = (n compose f) >=> (n compose g)
n compose pure = pure
```

where

```
(f >=> g)(a) = f(a) flatMap g
```

An “interpretation” of a monad M_1 into another monad M_2 will be a **monad morphism** $M_1 \to M_2$.

This is important: it means we can be sure our interpreted program continues to obey the monad laws.

Otherwise, we’ll lose predictability — the ability to reason about our code.

There's something interesting about vector spaces.

To define a linear function f from $\mathbb{R}^3$ to V, all we have to do is specify what f does to the elements of a special subset i.e. the basis $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$ (any basis will do).

The definition of f is automatic because of linearity:

$$f(x\mathbf{i} + y\mathbf{j} + z\mathbf{k}) = xf(\mathbf{i}) + yf(\mathbf{j}) + zf(\mathbf{k}))$$

We can write this as

$$\mathrm{Vect}(\mathbb{R}^3, W) \cong \mathrm{Set}(\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}, W)$$

We say that $\mathbb{R}^3$ is the *free* vector space on the set $\{\mathbf{i}, \mathbf{j}, \mathbf{k}\}$.

Actually **all** (finite-dimensional) **vector spaces** are free.

$$\mathrm{Vect}(\mathbb{R}S, W) \cong \mathrm{Set}(S, W)$$

Lists have a similar property.

To define a monoid morphism f from List$A$ to M, all we have to do is specify what f does to each element of A,

The definition of f is automatic, because a monoid morphism has to respect the monoid operation:

$$f(\text{List}(a_1, \ldots, a_k) = f(a_1,) \ldots f(a_k)$$

We say that List$[A]$ is the free monoid on A.

So now you understand what “free monad” means.

If Free F is the free monad on F, then to define a monad morphism n out of Free F to another monad N, all we have to do is “define what n does on F”.

In this case the relevant notion is a **natural transformation from F to N**.

$$\mathrm{Mon}(\mathrm{Free}\, F, N) \cong \mathrm{Nat}(F, N)$$

If F breaks up into a sum of functors

$$F = F_1 + \cdots + F_k,$$

then something else happens:

$$\begin{aligned} & \mathsf{Mon}(\mathsf{Free}\,(F_1 + \cdots + F_k), M) \\ \cong\;& \mathsf{Nat}(F_1 + \cdots + F_k, M) \\ \cong\;& \mathsf{Nat}(F_1, M) \times \cdots \times \mathsf{Nat}(F_k, M) \end{aligned}$$

In other words. . .

to interpret a program written in the free monad of a sum of languages $F_1, \ldots, F_k$

we just have to give a (functor) interpretation for each language.

So this gives us modular interpreters.

Example

```
case class Ask[A](s: String, next: String => A)
case class Tell[A](s: String, next: Unit => A)
case class Lookup[K, V, A](key: K, next: Option[V] => A)
```

We can already write programs already using these operations...

But they'd be a horrible mess of `Join`, `Pure`, `Inl`, `Inr` and type annotations.

And worse, if we want to reuse a program in an extended language, we'll have to go through and change all the 'Inl's and otherwise we would lose code reuse when we added a new language.

Boilerplate

We can fix this, but we'll need to define a few things first.

We want to capture the notion that for example `G` is "contained" in `F + G + H`.

We need this to remove the explicit `Inl` and `Inr`s.

So let's define a new (multi-parameter) type class:

```
sealed trait :<:[F[_], G[_]] {
  def inj[A](fa: F[A]): G[A]
}
```

and some instances

```
object :<: {
  implicit def refl[F[_]] = new (F :<: F) {
    def inj[A](fa: F[A]): F[A] = fa
  }
  implicit def left[F[_], G[_]] =
   new (F :<: Coproduct[F, G, ?]) {
    def inj[A](fa: F[A]): Coproduct[F, G, A] =
      Inl(fa)
   }
  implicit def right[F[_], G[_], H[_]]
   (implicit fh: F :<: H) =
   new (F :<: Coproduct[G, H, ?]) {
    def inj[A](fa: F[A]): Coproduct[G, H, A] =
      Inr(fh.inj(fa))
   }
}
```

This now lets us write a general inclusion (inject) function:

```
def inject[F[_], G[_], A](fga: F[Free[G, A]])
(implicit fg: F :<: G): Free[G, A] =
  Join(fg.inj(fga))
```

This will let us hide away the explicit `Join`s, `Inl`s and `Inr`s.

More boilerplate!

Rewrite our program in terms of smart constructors.

```
def ask[F[_]](s: String)
(implicit sub: Ask :<: F): Free[F, String] =
  inject(Ask[Free[F, String]](s, Pure(_)))
```

Invisible abstraction

Finally!

```
val checkQuota: Free[Ask + Lookup + Tell, Unit] =
  for {
    name <- ask("What's your name?")
    quota <- lookup(name)
    _ <- tell("Hi " + name + ", your quota is "
         + quota.getOrElse(0))
  } yield ()
```

Interpretation – more boilerplate

Introduce another type class.

```
class (Functor f, Monad m) => Interpret f where
  interp :: f a -> m a

implicit def interpretSum[F[_], G[_], M[_]]
  (implicit IF: Interpret[F, M], IG: Interpret[G, M])
  : Interpret[Coproduct[F,G,?], M] = ...

def interpret[F[_], M[_]](fa: Free[F, A])
  (implicit I: Interpret[F, M]): M[A] = ...
```

Note that we know `interpret` is a monad morphism.

```
case class MyMonad(run: Reader (Map[String Int])
implicit val monad: Monad[MyMonad] = ...

implicit val interpretAsk: Interpret[Ask, MyMonad] = ...
implicit val interpretTell: Interpret[Tell, MyMonad] = ...
implicit val interpretLookup:
  Interpret[Lookup[String, Int, ?], MyMonad] = ...
```

Let's run our program.

```
> run (run (interpret checkQuota)
  List("Stu" -> 33, "Ann" ->55)))
What's your name?
Ann
Hi Ann, your quota is 55
```

Specification-side effects

It's not clear how we could easily add effects like early termination to our program: we'd need to create a `FreeMonadError`.

Other binding constructs

- If we wanted to use comonads instead, we could use cofree comonads, but we'd have to roll our own "codata types à la carte".
- If we wanted arrows, it's even less clear what to do.

Although free arrows ought to exist, we don't yet have a Scala implementation.

Limitations

- It's slow.
- it's complex. We've had to add considerable boilerplate: injections, smart-constructors, interpreters.
- No backtracking in Scala's type solver means `:<:` can only nest on one side.

Solution 2: Operations remain functions

Oleg Kiselyov, Typed Tagless Final Interpreters, 2012

Motto

Let the language do the work

A language is a type class parametrized by a data constructor.
i.e. the parameter has kind `* -> *`.
i.e. a “higher-kinded” type

```
trait Console[R[_]] {
  def ask(s: String): R[String]
  def tell(s: String): R[Unit]
}

trait KeyVal[K, V, R[_]] {
  def lookup(key: KeyVal): R[Option[V]]
  def set(key: K, value: V): R[Unit]
}
```

An interpreter is a type with an instance.

```
class StdIO[A](run: => A)

object StdIO {
  implicit val console = new Console[StdIO] {
    def tell(s: String): StdIO[Unit] =
      new StdIO(print(s))
    def ask(s: String): StdIO[String] =
      new StdIO(io.StdIn.readLine(s))
  }
}

class NetworkConsole[A](address: URL, run: => A \/ Error)

object NetworkConsole {
  implicit val console = new Console[NetworkConsole] {
    def tell(s: String): NetworkConsole[Unit] = ...
    def ask(s: String): NetworkConsole[String] = ...
  }
}
```

```
implicit def mapKeyVal[K, V]: KeyVal[K,V, State[Map[K,V], ?
  new KeyVal[K, V, State[Map[K, V], ?]] {
    def lookup(key: K): State[Map[K, V], Option[V]] =
      State.gets(_.get(key))

    def set(key: K, value: V): State[Map[K, V], Unit] =
      State.modify(_ + (key -> value))
  }
```

Adding languages simply means adding type constraints.

It's easy to split languages.

```
trait Lookup[K, V, R[_]] {
 def lookup(key: K): R[Option[V]]
}

trait Set[K, V, R[_]] {
  def set(key: K, value: V): R[Unit]
}
```

. . . and to combine them. Just use the constraints together.

So our languages are modular.

We have modular interpreters just by adding new instances for our type.

For a monadic computation, just add a Monad constraint.

```
type StringIntKeyVal[R[_]] = KeyVal[String, Int, R]

def getQuota[R[_]](
  implicit C: Console[R],
           L: StringIntKeyVal[R],
           M: Monad[R]
  ): R[Unit] = {
  import C._, L._, M._

  for {
    name <- ask("What's your name?")
    quota <- lookup(name)
    _ <- tell("Hi " + name + ", your quota is "
      + quota.getOrElse(0))
  } yield ()
}
```

By defining syntax traits we can clean this up further:

```
def getQuota[R[_]:Console:StringIntKeyVal:Monad):R[Unit]=
  for {
    name  <- ask("What's your name?")
    quota <- lookup(name)
    _     <- tell("Hi " + name + ", your quota is "
             + quota.getOrElse(0))
  } yield ()
```

```scala
trait ConsoleSyntax {
  def ask[R[_]](s: String)(implicit R: Console[R]) =
    R.ask(s)
  def tell[R[_]](s: String)(implicit R: Console[R]) =
    R.tell(s)
}
```

Adding effects to the specification is easy.

Just add a different constraint in place of `Monad`.

```scala
def changePwd[R[_]:Console:StringStringKeyVal:MonadThrow]
  : R[Unit] =
  for {
    n       <- ask("What's your name?")
    p       <- ask("What's your password?")
    matches <- lookup(n).map(_ == Some(p))
    _       <- unlessM(matches)(throwM(WrongPassword))
    np      <- ask("Enter new password")
    np2     <- ask("Re-enter new password")
    _       <- unlessM(np == np2)
                (throwM(PasswordsDidNotMatch))
    _       <- set(n, np)
    _       <- tell("Password successfully updated")
  } yield ()
```

To have a comonadic or arrowized computation. . .

. . . just add the relevant constraint.

Interpretation is the identity function.

i.e. just select a type (that has an instance for all constraints).

Example

```scala
case class MyMonad(run: State[Map[String, String]])

implicit val console: Console[MyMonad] =
  ... // write to stdout, read from stdin

implicit val keyVal: StringStringKeyVal[MyMonad] = ...

implicit val monadThrow: MonadThrow[MyMonad] = ...
```

Let's run our program:

```
>  changePwd[MyMonad](
    Map(
      "anne" -> "pwd123",
      "mark" -> "p@ssw0rd"
    )
  ).run.run

What's your name?
mark
What's your password?
p@ssw0rd
Enter new password
1337
Re-enter new password
1337
((),List(("sue","pwd123"),("mark","1337")))
```

Comparison

Datatype à la carte

Free monads over sums of functors.

An interpreter is

- a monad morphism
- assembled in a modular fashion from interpretations (natural transformations) for each component language.

- Can’t use comonads or arrows.
- Can’t deal with effects in the specification.
- Needed quite a bit of boilerplate.
- Considerable runtime overhead.

Typed tagless

Languages are (higher-kinded) type classes.

We combine languages by adding type class constraints.

Interpretations are types with the necessary instances.

Compared to à la carte, the tagless approach

- has minimal runtime overhead
- needs less boilerplate

On the other hand, some things are more difficult e.g.

- stepping through instruction by instruction
- translating to another language

but not impossible (see the “Typed tagless final” paper).
Reusing interpreters can be more of an art than a science — there are several different approaches.

Recommendation

Prefer typed tagless — faster, less fuss.

But in either case, what we've developed is a whole lot more powerful and less intrusive than a dependency injection framework.

Related work

- Stacked `FreeT`.
- Alternative (more efficient) implementations of free monads
  e.g. van Laarhooven free monad, “reflection without remorse”
- Algebraic effects — avoid a monad stack altogether

  e.g. "Freer monads, more extensible effects"
- Extensible data types

  see the “Data types à la carte’ and “Typed tagless final interpreters” papers.

References

Swierstra, Wouter. “Data types à la carte.” Journal of functional programming 18.04 (2008): 423-436.

Bahr, Patrick, and Hvitved, Tom. “Compositional data types.” Proceedings of the seventh ACM SIGPLAN workshop on Generic programming. ACM, 2011.

Kiselyov, Oleg. “Typed tagless final interpreters.” Generic and Indexed Programming. Springer Berlin Heidelberg, 2012. 130-174.