Dependency Injection Without the Gymnastics

Functional Programming Applied

Tony Morris

Philadelphia Emerging Technologies for the Enterprise — April 10-11, 2012

In this talk, we take a look at some specific Functional Programming patterns that regularly arise in everyday programming. These patterns are used to solve the same problems that DI attempts to by decoupling data types from their dependencies.

We explore the algebraic properties of these techniques and discover how they give rise to useful programming properties that we may apply in our typical work. This talk focuses on the techniques rather than any particular programming language, since none of these techniques are monopolized by a specific language. Various programming languages will be used for demonstration to emphasize this point; Haskell, Scala, Java and C# for example.

The audience should expect to walk away from this talk with a slight amount of bewilderment, a clear understanding of some FP subjects, but importantly, with the invigorated inspiration to explore these techniques further.

Table of Contents

Introduction

- Runar went deep into the Reader monad to discuss dependency injection[1]
- I am going to fly-by the Reader monad, then enter a rapid climb, levelling out at the `ReaderWriterStateT` monad transformer.
- I am going to ask you to hold a *lot more* in your head, inspiring you to follow it up with applications later.
- We will use mostly Scala for code listings.
- You may get a little uncomfortable. This is a *good thing*.

The Real Goal

The real goal is:

- to introduce you to some interesting and useful data structures
- to introduce you to a not-so-beginner view of a programming technique (I like living on the edge — do you?) called *pure functional programming*
- to achieve these two goals by appealing to a problem you are likely already familiar with (DI)
- it's going to look a bit like we are picking on DI, though this is not a goal (it does a spectacular job of that on its own)

What is Dependency Injection?

The Essence of DI

- Dependency injection often comes in two general forms:
  1. Applications require a read-only *context*, which is initialised before the dependent application starts.
  2. Applications require a read/write *context*, which is initialised before the dependent application starts.
- In both cases, the DI runtime guarantees that the context initialisation will occur at the right time — this is the essential promise of dependency injection.[2]
- The rest is uninteresting ceremony.

[1] Dead-Simple Dependency Injection http://lanyrd.com/2012/nescala/sqygc/
[2] Dependency injection is just socially-acceptable global variables — Anon

Gymnastics

- ‹Personally/›, I have seen *all sorts* of ‹gymnastics sorts="all"/› in an effort to provide this guarantee.
- After all, if this guarantee is broken at any time, the program **crashes spectacularly**.
- Let's be honest, it never works consistently — it's just a global variable with all the usual dangers.
- OK then, pass arguments through explicitly — bit clumsy innit?

Generalising Programs

A Trivial Scala Program

```
val a = e1
val b = e2(a)
val c = e3(a, b)
val d = e2(b)
```

- Remove the `val` keyword.
- Replace the = symbol with `<-`
- Wrap the program in `for` and `yield` keywords.

A Trivial Program in Identity

```
for {
  a <- e1
  b <- e2(a)
  c <- e3(a, b)
  d <- e2(b)
} yield d
```

We can transform our programs like this by wrapping values (`e1`, etc.) in `Identity`.

```
case class Identity[A](a: A) {
  def map[B](f: A => B): Identity[B] = Identity(f(a))
  def flatMap[B](f: A => Identity[B]): Identity[B] = f(a)
}
```

Add a Bit of Context

- OK, so if `e1` was of the type `Int`, it now becomes `Identity[Int]` and so on.
- but `Identity` is quite a boring context.
- Are there more structures that add more interesting context to values of any type?

Reader

```
case class Reader[A](rd: Context => A) {
  def map[B](f: A => B): Reader[B] =
      Reader(f compose rd)
  def flatMap[B](f: A => Reader[B]): Reader[B] =
```

```
      Reader(c => f(rd(c)) rd c)
}
```

- Now values that were once typed `T` becomes values with the type `Reader[T]`.
- In other words, these values can access a `Context` to produce their result.
- This concept of "accepting an argument" is often referred to as *Inversion of Control*.
- …and most importantly…

Reader Monad

```
for {
  a <- e1
  b <- e2(a)
  c <- e3(a, b)
  d <- e2(b)
} yield d
```

- …the program remains unchanged.
- Here, we are computing values with a context available, but *without explicitly passing it through our program*.

Example

A simple enough example?

- Can we find an example good enough to demonstrate the point, but small enough to fit on slides?! :)
- We can do this *for any value* in our context.
- Let our context be a hostname and port pair, `(String, Int)`.
- We are going to *inject* this read-only dependency into our program, so that:
  - values that require access to it can do so
  - the dependency is not explicitly passed around our program
  - our program maintains *referential transparency — no variables*, and therefore, the benefits that follow

Set up Libraries

- ```
  // Hostname+port reader
  case class CReader[A](rd: (String, Int) => A) {
    def map(f: A => B): CReader[B] =
      sys.error("todo")
    def flatMap(f: A => CReader[B]): CReader[B] =
      sys.error("todo")
  }
  ```

- ```
  // Ignore the context, lift into CReader
  def point[A](a: => A): CReader[A] =
      CReader(_ => a)
  ```

Usage example

```
for {
  _ <- log("connect: %s:%s")
  w <- getWibble
  val v = w modL biggen
  u <- setWibble(v)
  _ <- log("disconnect %s:%s")
} yield u
```

Some important points of note:

- `log: String => CReader[X]` with a catch!
- `getWibble/setWibble: CReader[Wibble]`
- `modL` is a *lens operator* for working with immutable records such as `Wibbles`.

...and it's not all that different to our everyday programming

Example 1. The same code in C#/LINQ using monad comprehension syntax

```
from a in Log("connect: %s:%s")
from w in GetWibble
from u in SetWibble(w.ModL(Biggen))
from b in Log("disconnect %s:%s")
select u
```

- What Scala calls:
  - `flatMap`, C# calls `SelectMany`
  - `map`, C# calls `Select`

...and in Haskell

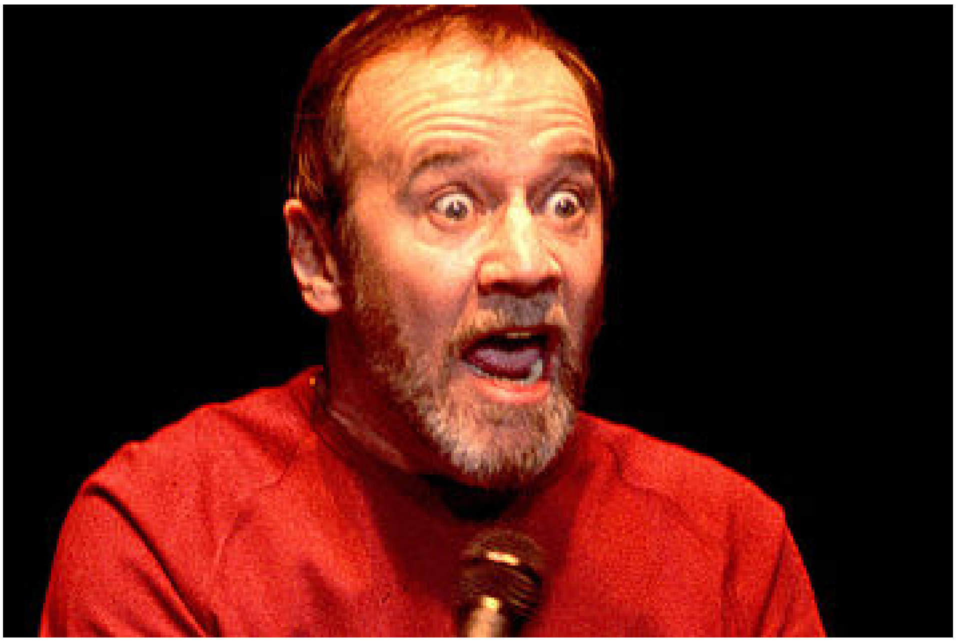
Example 2. Using Haskell monad comprehension syntax

```
do log "connect: %s:%s"
   w <- getWibble
   let v = w modL biggen
   u <- setWibble v
   log "disconnect %s:%s"
   return u
```

State

But then the TPS report comes in from the senior enterprise architellij...

... and the product manager wants the ability to allow the user to dynamically synergise the port number on the fly



"Oh no! More stuff!" — George Carlin

Injecting a Read/Write Context

- We have seen a read-only Context (called Reader) over a value (A)

  `Context => A`

  - is a functor — has a useful `map` method
  - is a monad — has a useful `flatMap` method

- Read/Write Context (called State) over a value (A)

  `Context => (A, Context)`

State Monad

`State` is a monad too!

```
case class State[Cx, A](run: Cx => (A, Cx)) {
  def map[B](f: A => B): State[Cx, B] =
      State(x => {
        val (a, t) = run(x)
        (f(a), t)
      })
  def flatMap[B](f: A => State[Cx, B]): State[Cx, B] =
      State(x => {
        val (a, t) = run(x)
        f(a) run t
      })
}
```

Better fix the program then innit?

```
for {
  _ <- log("connect: %s:%s")
```

```
  w <- getWibble
  val v = w modL biggen
  u <- setWibble(v)
  _ <- log("disconnect %s:%s")
} yield u
```

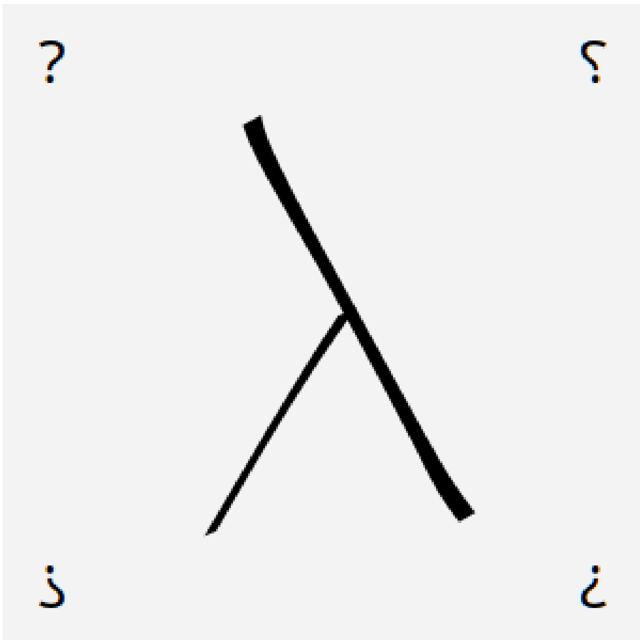
- The program remains **unchanged** due to generalisation.
- However, now we can add our context-writes *without introducing in-place variables.*

Add a Context Write

We can change the port as the program runs, because we can change to a read-write context easily and without affecting orthogonal program parts.

```
for {
  _ <- log("connect: %s:%s")
  _ <- setPort(80) // context-write
  w <- getWibble
  val v = w modL biggen
  u <- setWibble(v)
  _ <- log("disconnect %s:%s")
} yield u
```

Hammer Time



Now is a good time to pause for questions.

Contexts All the Way Down

Transformers

- We have seen read-only contexts and read-write contexts, but what about others?
  - Nullability (`Option`)
  - Non-determinism (`List`)

    - Logging (`Writer`)
    - I/O
    - Exception handling (`Either[Throwable, _]`)
- Importantly, what if we want to *combine* two contexts to create a new context?
    - We might want to read a context and compute a nullable (`Option`) value. Can we combine these two contexts?

ReaderT

- Was:

```
case class Reader[A](rd: Context => A)
```

- Now:

```
case class ReaderT[F[_], A](rd: Context => F[A])
```

    - `ReaderT` has a `map` method if `F` has a `map` method
    - `ReaderT` has a `flatMap` method if `F` has a `flatMap` method
    - `type Reader[A] = ReaderT[Identity, A]`

Stacking It Up

- Remember those `log` statements? [1] [2] [3] They have `F=Writer` stacked in the context — just another context!
- Not only are we injecting with composition, but injecting into a stack of contexts that can be chosen specific to the problem at hand.
- We can do it with read-write contexts (`State`) too.

```
case class StateT[Cx, F[_], A](run: Cx => F[(Cx, A)])
type State[Cx, A] = StateT[Cx, Identity, A]
```

… wait a minute … if `Option` has `flatMap` …

```
case class OptionState[Cx, F[_], A](run: StateT[Cx, Option, A])
```

- Again, our program client remains unchanged, running through its for-comprehension.
- Reader has `flatMap` too!

```
case class ReaderState[F[_], A](run: StateT[Cx, Reader, A])
```

- We can stack *indefinitely*, thus injecting dependencies arbitrarily and with solid library support.

Sceptical?

- You are now probably one of three people:

- WTF is this guy smokin'? Give me my XML, variables, bugs, frameworks and really big JAR files back right now!
- I wonder what you just said; can I see some code now?
- *yawn* seen it all before, been doing it for years, when is lunch?

- To understand this subject more deeply, there are plenty of resources and people to help — just ask!

Reader/Writer/State Transformer

The Essence of Injecting Dependencies

Putting it All Together

- Dependency Injection is, quite concisely, a specialisation of the reader/writer/state monad transformer.
- The *what what* transformer?
- The transformer that is stacked with `Reader`, `Writer`, `State` and an arbitrary context

Reminder

- `case class Reader[R, A](f: R => A)`
- `case class Writer[W, A](w: (W, A))`
- `case class State[S, A](f: S => (A, S))`
- Reader+Writer+State?

RWS Transformer in Scala

```
case class ReaderWriterStateT[R, W, S, F[_], A](
  run: (R, S) => F[(W, A, S)]
) {
  def map[B](f: A => B)(implicit F: Functor[F])
  : ReaderWriterStateT[R, W, S, F, B] =
    ReaderWriterStateT {
      case (r, s) => F.map(run(r, s)) {
        case (w, a, s) => (w, f(a), s)
      }
    }

  def flatMap[B](f: A => ReaderWriterStateT[R, W, S, F, B])
                (implicit F: FlatMap[F], W: Semigroup[W])
  : ReaderWriterStateT[R, W, S, F, B] =
    ReaderWriterStateT {
      case (r, s) => F.flatMap(run(r, s)) {
        case (w1, a, s1) => F.map(f(a) run (r, s1)) {
          case (w2, b, s2) => (W.op(w1, w2), b, s2)
        }
      }
    }
}

object ReaderWriterStateT {
  type ReaderWriterState[R, W, S, A] =
```

```
  ReaderWriterStateT[R, W, S, Id, A]
}

case class ReaderT[A, F[_], B](
                                  rd: A => F[B]
                                  ) {
  def rws[W, S](implicit F: Functor[F], W: Monoid[W])
  : ReaderWriterStateT[A, W, S, F, B] =
    ReaderWriterStateT {
      case (r, s) => F.map(rd(r))(
        (W.id, _, s)
      )
    }
}

object ReaderT {
  type Reader[A, B] =
  ReaderT[A, Id, B]
}

case class WriterT[W, F[_], A](
                                  wx: F[(W, A)]
                                  ) {
  def rws[R, S](implicit F: Functor[F])
  : ReaderWriterStateT[R, W, S, F, A] =
    ReaderWriterStateT {
      case (r, s) => F.map(wx){
        case (w, a) => (w, a, s)
      }
    }
}

object WriterT {
  type Writer[W, A] =
  WriterT[W, Id, A]
}

case class StateT[S, F[_], A](
                                 st: S => F[(A, S)]
                                 ) {
  def rws[W, R](implicit F: Functor[F], W: Monoid[W])
  : ReaderWriterStateT[R, W, S, F, A] =
    ReaderWriterStateT {
      case (r, s) => F.map(st(s)){
        case (a, s) => (W.id, a, s)
      }
    }
}

object StateT {
  type State[S, A] =
  StateT[S, Id, A]
}

case class Id[A](a: A)

object Id {
  implicit val IdFlatMap: FlatMap[Id] =
```

```
    new FlatMap[Id] {
      def fmap[A, B](f: A => B) =
        i => Id(f(i.a))
      def bind[A, B](f: A => Id[B]) =
        i => f(i.a)
    }
}

trait Functor[F[_]] {
  def fmap[A, B](f: A => B): F[A] => F[B]
  def map[A, B](a: F[A])(f: A => B): F[B] =
    fmap(f)(a)
}

trait FlatMap[F[_]] extends Functor[F] {
  def bind[A, B](f: A => F[B]): F[A] => F[B]
  def flatMap[A, B](a: F[A])(f: A => F[B]): F[B] =
    bind(f)(a)
}

trait Semigroup[A] {
  def op(a1: A, a2: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def id: A
}
```

RWS Transformer in Haskell

```
newtype ReaderWriterStateT r w s f a =
  ReaderWriterStateT {
    run :: (r, s) -> f (w, a, s)
  }

instance Functor f =>
  Functor (ReaderWriterStateT r w s f) where
  fmap f (ReaderWriterStateT x) =
    ReaderWriterStateT $
      fmap (\(w, a, s) -> (w, f a, s)) . x

instance (FlatMap f, Semigroup w) =>
  FlatMap (ReaderWriterStateT r w s f) where
  flatMap f (ReaderWriterStateT x) =
    ReaderWriterStateT $ \(r, s) ->
      flatMap (\(w1, a, s1) ->
        fmap (\(w2, b, s2) ->
          (op w1 w2, b, s2))
          (run (f a) (r, s1))) (x (r, s))

type ReaderWriterState r w s a =
  ReaderWriterStateT r w s Id a

newtype ReaderT a f b =
  ReaderT { rd :: a -> f b }

rwsR (ReaderT f) =
  ReaderWriterStateT $ \(r, s) ->
```

```
    fmap (\z -> (identity, z, s)) (f r)

type Reader a b =
  ReaderT a Id b

newtype WriterT w f a =
  WriterT { wx :: f (w, a) }

rwsW (WriterT x) =
  ReaderWriterStateT $ \(r, s) ->
    fmap (\(w, a) -> (w, a, s)) x

type Writer w a =
  WriterT w Id a

newtype StateT s f a =
  StateT { st :: s -> f (a, s) }

rwsS (StateT f) =
  ReaderWriterStateT $ \(r, s) ->
    fmap (\(a, s) -> (identity, a, s)) (f s)

type State s a =
  StateT s Id a

newtype Id a = Id a

instance Functor Id where
  fmap f (Id a) = Id (f a)

instance FlatMap Id where
  flatMap f (Id a) = f a

class Functor f => FlatMap f where
  flatMap :: (a -> f b) -> f a -> f b

class Semigroup a where
  op :: a -> a -> a

class Semigroup a => Monoid a where
  identity :: a
```

A. Tony Morris — PGP Key


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.6 (GNU/Linux)

mQGiBETNyC0RBAC3MYSZSbDZhBLKra2YUphB9OO6+qMFl/v2Lq8590ZfeE2WjIOu
c/KGKyOigXztMrA4+iekUjM4FA8E6AlBRQiAqZK8HF0ftX5hDpuSyEKkZe3jcxxI
BbhwX/SWHtDEVzwNmvO1wEnwhRE1oY/BCmy+bQ9wmAjlNav4UbOIcXcJUwCgz6FF
UwDvPxzybgd9FNS14BE37n8D/AzGmGjunBW/x+g/ndwu6WEpTEn1ZNdja6VrNXSG
xQ8XM+NBulroAIYX+YWdHsKnuGvSKCgVoc1ifVHdztA9sksID5GBGzhmVbIP2E16
w/LEzqqwruv9dX0Y1b7n8hcnvbl4DgEgLgeQ+VgJfLUkY2jFZ2m3dEBRH9USSgB/
V2pBBACP4Us2ZBYEXtMG29g6GqyeeJLEP34PLYVHWJZbet/wPQsHFhYahhzjwZGG
Srp0JUpegJOdaqX/Y0nio2whgCpqJcrbGuUBqNgQI3k4gvBwnPyx7jM0kfHfFORG
lq9SDbfLpV0EJx6fWXGStW33zsQQvenDcV2F5czzKQcy66BFwbQhVG9ueSBNb3Jy
aXMgPHRtb3JyaXNAdG1vcnJpcy5uZXQ+iGMEExECACMFAkTNyC0FCQlmAYAGCwkI
```

```
BwMCBBUCCAMEFgIDAQIeAQIXgAAKCRCaemCth7qvrf0pAKCFii2pI2W1BKVFuQcw
yoNxP0CAUACgyhuI9isCvtrOkeyjDmCVueRCdaC5Ag0ERM3IThAIAJ6A1z+d40ve
WvPIhpFiGfoS8UX4YdgYpo2mC/orY0xszBitogtaTHQHU5YDemGg81plNg9I3DbM
Er8uyONV4DqF6RbLj4w+iA6zn93+PTEZ73ydhxF6vDuojpVZPXVzXzpgyXHkEVLC
3hKL9oVlEsh+DWCvCiSAIy780JZ3FNVuMC3VH4qKxTw0CwPuuZvVfnMoIRfpODRR
fVEk2VDor+lr8kqJkBaHgN5o/AvOXC7QCYadwbEkpr0ecxIZ1VcASYytIIM3YNL7
ZcHWwU5PCNLOdMXPqOdthhDhsHkKJNEXXr0YsjX/bQqYOUqYKPDyqh/yrrRO9Ro6
7eTSbfIguycAAwcH/2waLIQR8qYKxPknNuSdsOOqF2jf2gglL/7uMsIzjfkFzgHo
+GNHw9tmlZqD3yzaZ/N7Yv08ujRHhmWPBYAWRICBM3qo0zMJ9kI5XWRobeRQpLtf
YxxIOenq8R9t6YU9ryHdqf+P+Fi38eN5ERTDhNLrJOnO5/TA+of97BWCmdtJMlWM
RaHtqXxwo02Yi65IqKx6L7oOvT7Gh4NV2eglz8ZafPEoP8+V8ER7rwBYPiLk4Mse
oVImjveq2dmLUip9OPwznoaeyC8zB0mJ13m/KNC+CffkBgoXpMPiKzbu5YTjVw++
MlEmfgL42yDK0hnokmW2i9y1RBh/T1VQQeQbUaeITAQYEQIADAUCRM3ITgUJCWYB
gAAKCRCaemCth7qvrcbtAJ9j3C6lKNRB3uKcrfze66jAVQh0qACaAysOK82TcQ/2
73ryR0xWMFnpGqg=
=bMTb
-----END PGP PUBLIC KEY BLOCK-----
```


B. Licence

   4. "Distribute" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.

   5. "License Elements" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.

   6. "Licensor" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

   7. "Original Author" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

   8. "Work" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

   9. "You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

   10. "Publicly Perform" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

   11. "Reproduce" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights. Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

   1. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;

2. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";

3. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,

4. to Distribute and Publicly Perform Adaptations.

5. For the avoidance of doubt:

    1. Non-waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

    2. Waivable Compulsory License Schemes. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,

    3. Voluntary License Schemes. The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

    1. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.

    2. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply