The categorical "flavour" of the Reactive Extensions (aka Rx)

and how it can help us building better reactive code

Alessandro Candolini

January 25, 2018

Erik Meijer (father of Rx)

Agenda

1. Naive introduction to Rx
2. Categorical duality
3. Quick (and mostly incomplete) tour about monads
4. Leaving the monad

Naive introduction to Rx

```
/* Observable (data stream) */
val stream: Observable<String> =
            Observable.just("wejd", "adheui", "fe")


/* Observer */
stream.subscribe(
    { s -> println(s) }, /* onNext */
    { t -> println(t) }, /* onError */
    { println("Completed") } /* onComplete */
  )
```

```kotlin
val stream: Observable<String> =
            Observable.just("wejd", "adheui", "fe")

stream.map { s -> s.length } //<--
    .filter { l -> l > 2 } //<--
    .subscribe(
       { l -> println(l) },
       { t -> println(t.localizedMessage) },
       { println("Completed") }
    )
```

```kotlin
val stream: Observable<String> =
        Observable.just("wejd", "adheui", "fe")

stream.map { s -> s.length }
    .filter { l -> l > 2 }
    .subscribeOn(Schedulers.io()) //<--
    .observeOn(AndroidSchedulers.mainThread()) //<--
    .subscribe(
       { l -> println(l) },
       { t -> println(t.localizedMessage) },
       { println("Completed") }
    )
```

```kotlin
val stream: Observable<String> =
            Observable.just("wejd", "adheui", "fe")

val integers = stream.map { s -> s.length }
      .filter { l -> l > 2 }

integers.subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
           { l -> println(l) },
           { t -> println(t.localizedMessage) },
           { println("Completed") }
         )
```

```kotlin
/** Data source */
val stream: Observable<String> =
            Observable.just("wejd", "adheui", "fe")

/** Business logic */
val integers = stream.map { s -> s.length }
      .filter { l -> l > 2 }

/** Consumer */
integers.subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(
           { l -> println(l) },
           { t -> println(t.localizedMessage) },
           { println("Completed") }
         )
```

```
data class User(name : String)

interface Repository {
    fun fetchUser(): Observable<User> /*Single??*/
}

repository.fetchUser()
     .map { user -> user.name }
     .subscribeOn(Schedulers.io())
     .observeOn(AndroidSchedulers.mainThread())
     .subscribe(
         { name -> /* Show name */ },
         { t -> /* Show error */  },
         { println("Completed") }
     )
```

```kotlin
/* Data source */
val user = repository.fetchUser()

/* Business logic */
val name = user.map { user -> user.name }

/* Consumer layer */
names.subscribeOn(Schedulers.io())
     .observeOn(AndroidSchedulers.mainThread())
     .subscribe(
         { name -> /* Show name */ },
         { t -> /* Show error */  },
         { println("Completed") }
     )
```

Naively speaking, Rx provides way to:

- create observables backed by either static or dynamic data
    - lists, sensors data, network or db queries, etc
- manipulate and combine streams
    - dot-chaining, built-in operators, etc
- observe the stream of events

In addition, Rx provides:

- propagation of errors along the chain
- declarative multithreading
    - parametrize concurrency using schedulers
    - abstract away low-level threading

Caution

- It's *not* an *event bus* (more on this coming soon)

- It's not a library for (just) abstract away low-level threading

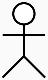
- observables are *push-based* (i. e., reactive)
    - exploit *duality* (category theory) to *build* observables from iterables
- monadic behavior of observables
    - focus on “happy path“, propagate exceptions all the way down to the observer
- Provide abstraction on top of asynchronous vs synchronous behavior
    - Rx is parametrised over *concurrency* (i. e., schedulers), not over time
    - Inversion of control: we can delegate to the consumer the choice of the schedulers
- *Not just FP*: it does support *side effects* (doOnSomething, Subjects, etc)

Categorical duality

Iterable	Observable
pull-based	push-based
interactive	reactive
consumer is in charge	producers is in charge
consumer is active	consumer is passive
blocking	non-blocking
no backpressure	backpressure
backed by static or dynamic data	backed by static or dynamic data

```
interface Iterable<out T> {
  fun iterator(): Iterator<T>
}
```

```
interface Iterator<out T> {
  fun hasNext(): Boolean
  fun next(): T //<--NoSuchElement
}
```



Active consumer

Let's swap arguments and results (mathematical duality).

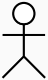
```
interface Iterable_<out T> {
    // setter
    fun set(iterator_: Iterator_<T>) : Unit
}

interface Iterator_<in T> {
   fun onHasNext(hasNext : Boolean)
   fun onNext(t: T)
   fun onError(error: Throwable)
}
```

Code cleanup:

```
interface ObservableSource<out T> {
    fun subscribe(observer: Observer<T>) : Unit
}
```

```
interface Observer<in T> {
  fun onNext(t: T)
  fun onError(e: Throwable)
  fun onComplete()
}
```



Passive consumer

Iterables are pull based: it sits until someone asks for `next()` value, you can *iterate* and pull values

Observables are pushed based: they decide when to emit values, they *notify* when a new value is pushed

> *"Earthquakes, in fact, are a really good example of a push based stream. We're not polling the earth for earthquakes."*

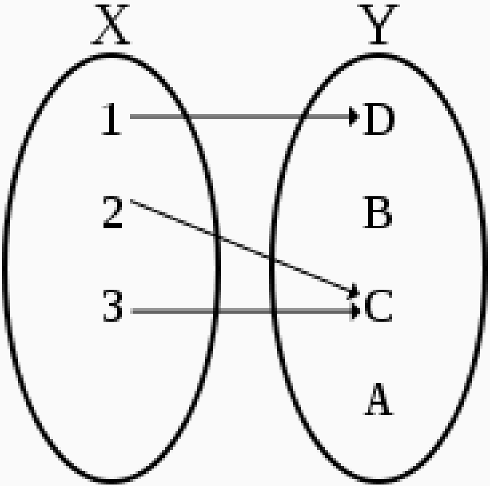
Erik Meijer

Other implications of duality:

- I can transport results proven for one system into dual results valid for the other system
- Duality can be used to achieve *inversion of control*

Quick (and mostly incomplete) tour about monads

Mathematical functions



FP “keywords”:

- pure functions
- no side effects
- immutability
- referential transparency

Implications (among other things):

- no loops
- no if as a control flow (only as a statement)
- no sequence of instructions: just function composition

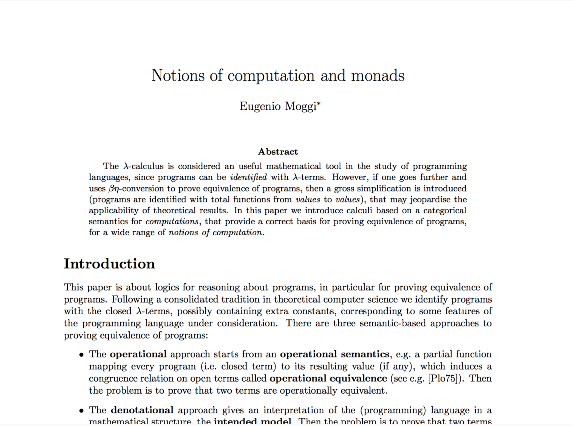
Weapons in the FP toolkit:

- Recursion
- Patterns for immutable collections: map, filter, reduce (we can't use loops)
- Algebraic data types & Pattern matching
- Lazyness
- And more...

But this is not enough.

Monads can be used

- as a container
- to structure computations in terms of values and sequences of computations
- to deal with side-effects in a purely functional setting (IO/ asynchronous behaviour, etc)
- to carry extra data (aka, state) and propagate it through the computation steps seemlessly
- and many more (the power of a good abstraction)



Notions of computation and monads

Eugenio Moggi*

Abstract

The λ-calculus is considered an useful mathematical tool in the study of programming languages, since programs can be *identified* with λ-terms. However, if one goes further and uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification is introduced (programs are identified with total functions from *values* to *values*), that may jeopardise the applicability of theoretical results. In this paper we introduce calculi based on a categorical semantics for *computations*, that provide a correct basis for proving equivalence of programs, for a wide range of *notions of computation*.

Introduction

This paper is about logics for reasoning about programs, in particular for proving equivalence of programs. Following a consolidated tradition in theoretical computer science we identify programs with the closed λ-terms, possibly containing extra constants, corresponding to some features of the programming language under consideration. There are three semantic-based approaches to proving equivalence of programs:

- The **operational** approach starts from an **operational semantics**, e.g. a partial function mapping every program (i.e. closed term) to its resulting value (if any), which induces a congruence relation on open terms called **operational equivalence** (see e.g. [Plo75]). Then the problem is to prove that two terms are operationally equivalent.
- The **denotational** approach gives an interpretation of the (programming) language in a mathematical structure, the **intended model**. Then the problem is to prove that two terms

Figure 1: E. Moggi, Information and Computation Volume 93, Issue 1, July 1991, Pages 55-92

Figure 2: Philip Wadler (haskell, monads, java generics, and more)

What's a monad?

> *All told, a monad in X is just a monoid in the category of endofunctors of X, with product replaced by composition of endofunctors and unit set by the identity endofunctor.*

Saunders Mac Lane in Categories for the Working Mathematician

> *Monads are return types that guide you through the happy path.*

Erik Meijer

> *Monads are parametric types with two operations flatMap and unit that obey some algebraic laws.*

Martin Odersky

Example (Maybe monad)

```
fun f( x : Int ) : Int = x * x
fun g( x : Int, y : Int ) : Int = x/y

val result = f( g( 2,3))
val result = f( g( 2,0)) ??
```

Java would naively solve this by using

- infamous `null`...
- exceptions

Example (Maybe monad continuation)

```
sealed class Maybe<out T> {
    object None : Maybe<Nothing>()
    data class Just<T>(val t: T) : Maybe<T>()
}
```

How to use this?

```
val j = Maybe.Just(1)    /* Wrap */
val (i) = j              /* Unwrap */
```

(sealed classes as kind of ADT, `when` as a kind of pattern matching)

In Haskell, more terse syntax. . .

```
data Maybe a = Just a | Nothing
```

Example (Maybe monad continuation)

```
fun g( x : Int, y : Int ) : Maybe<Int>
        = if ( y != 0 ) Maybe.Just(x/y)
            else Maybe.None
```

Troubles: *composition broken*

```
val intermediate = g(2,0)
when(intermediate){
  is Maybe.None -> {
     Maybe.None
  }
  is Maybe.Just<Int> -> {
     val (result) = intermediate
     f(result)
  }
}
```

Kleisli triple construction of a monad:

boxing (unit function): (identity, return, unit):

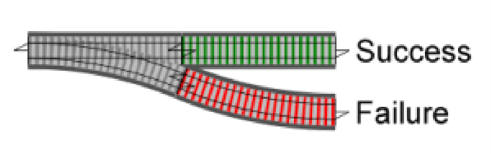
- Takes a value from a plain type and puts it into a container using the constructor, creating a monadic value

unboxing (binding): (bind, flatMap):

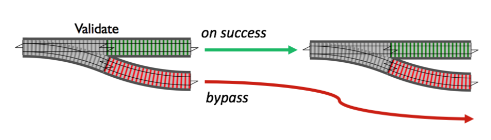
- The bind operator unwraps the plain value with type a embedded in its input monadic value with type M a, and feeds it to the function.

Unboxing allows to “connect” / “compose” / “link” sequence of functions calls by chaining several bind operators together

Railway driven development

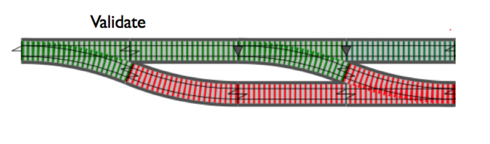


Railway driven development



Obviously, we cannot compose these two because of the input/output mismatch.

Railway driven development



Leaving the monad

```
repository.products()
    .doOnNext { products -> println(products) }
    .doOnSubscribe { /* do something*/ }
    .doFinally { /* do something*/ }
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
    { products -> /* Show products */ },
    { t -> /* Show error */  },
    { println("Completed") }
    )
```

```kotlin
class ProductUpdatesManager {

  private val publishSubject
        = PublishSubject.create<Product>()

  override fun update(product:Product) {
    publishSubject.onNext(product)
  }

  override fun changes() = publishSubject
}
```

Avoid side effects when possible

Rx side effects are not different from imperative control flows: they are error prone, inherently difficult to do concurency/asynchronous behaviour, problems with state management, etc (Example: loading spinner)

Subjects are antipatterns:

- Erik Meijer does not like Subjects[1] :)
- Jake Wharton does not like Subjects either[2]
- Subjects $\sim$ non-local side effects
- Subjects $\sim$ "global" state, stateful
- Subjects implies *fake decoupling*: you think you are decoupling the code (because your subscriber does not now the origin of the values emitted), but actually it means that your *observable source it's not bound*, it can emit everything from everywhere, and this think that you can suddenly receive a new event emitted and you dont know why,
- Ultimately, subjects $\sim$ event bus / broadcasting, so same drawbacks

[1] `https://social.msdn.microsoft.com/Forums/en-US/bbf87eea-6a17-4920-96d7-2131e397a234/why-does-emeijer-not-like-subjects`

[2] `https://github.com/JakeWharton/RxRelay/issues/7`

Bridge with imperative code can lead to antipatterns

```
class WrongPresenter : Presenter {
  fun onButtonClicked() {
     repository.fetch()
               .subscribe(
                 { product -> },
                 { error -> },
                 { }
               )
  }
}
```

```
repository.fetch().subscribe(
    { product -> },
    { error -> when(error) {
            is HttpsURLConnection -> { }
            is SSLException -> { }
            is SQLException -> { }
            else -> { }
        }
    }, { })
```

> *Anytime you find yourself writing code of the form if the object is of type T1, then do something, but if its of type T2, then do something else, slap yourself.*

Scott Meyers (Effective C++)

Questions?