Why on earth a ZIO IO monad for Scala?

1. Introduction.

In a recent article about the migration of a REST-API implementation from Akka to ZIO we made a statement along the following terms:

Scala being a hybrid FP-OO language, many Scala developers are used to intermix side-effecting code with pure functional code, thus writing impure functions that lack the property of referential transparency, which is at the base of i) safe function composition, the foundation for writing correct and maintainable functional programs, and ii) equational reasoning, a simple and powerful way to think about functional programs in order to evaluate them, prove their correctness, design them, transform them, etc.

What can that mean in practical terms? Well, it turns out that composition of pure functions and equational reasoning are a couple of very important things in practice that we lose due to impurity. But what are those things and why are so important? In this article we will try to answer that question by discussing some practical examples. In doing so, the notion of I/O monad will emerge and some aspects of ZIO as an I/O monad for console applications will be discussed.

> We will also mention in passing some formal concepts related to our discussion as asides in this format, directed to the more theory-inclined readers with the intention of motivating (or dissuading) them from doing more research on their part.

This article explains in Section 1 the notions of pure function, side effect and I/O monad, using informal definitions and examples. Section 3 offers an example of the application of those concepts to the development of a side-effecting program using ZIO as an I/O monad for console applications. To better understand that example, a very brief introduction to ZIO is presented in Section 2. The program developed in Section 3 implements a simple type of interactive dialogues, clearly separating their conceptual definition from their implementation as a console application. Section 4 discusses the advantages and further implications of that separation. Section 5 concludes with some remarks and a few references to additional material about I/O monads and ZIO.

1.1 Definition of pure functions and the most general concept of side effects.

A pure function in programming returns values that only depend on the values of their arguments, discarding any possibility of "changing behavior" from one call to the function to a subsequent one with the same arguments.

In other words, a pure function in programming implements the definition of a function in mathematics. This notion is expressed, in more operative terms, saying that pure functions in programming are those that are free of "side effects".

> Strictly speaking, to abide to the mathematical definition, functions in programming must also be total, i.e. they must be defined (return a value) for all the elements of their domains. Scala has the (otherwise very useful) notion

> of a partial function, in addition, of course, to including function that are intrinsically not total, such as arithmetic division which is not defined when the denominator is equal to zero. Scala partial functions are outside the scope of this article. Regarding "standard" Scala functions of type `X => A` which are undefined for some elements of `X`, they can be redefined as total functions by simply changing their type to `X => Option[A]` and returning `None` for those elements of `X` (and the same return value as before, wrapped by `Some`, for the rest).

In programming not all functions are pure. Let's give some examples of impure functions in Scala:

```
val rnd = new scala.util.Random

def addRandom(i:Int): Int = {
  val rndInt = rnd.nextInt
  i + rndInt
}

def getLine(): String = {
  println("Please type a line")
  val line = readLine()
  line
}
```

In both examples, two successive calls to the defined functions with the same arguments will return different values, so both functions are impure. In the first example, the source of impurity is an external service (specifically a random number generator initialized outside the scope of the function), in the second, an I/O console operation (actually two of them, but one would have sufficed). In both cases, the value returned by the impure service or operation depends on some external state (the state of the random number generator in the first case, the state of the console in the second), something that we could call "the state of the world".

This observation could give us another vivid colloquial definition of a side effect: something that interacts with the state of the world, either using it as input or modifying it (or both things). Impurity and state are inextricably linked, so much so that we could even characterize pure functions as those that return values that are not state-dependent.

Reversing our second, operative, definition of a pure function, we can also say that a side effect is anything that can break the "purity" of a function, anything that can cause a programming function to change behavior from one call to another with the same arguments. In other terms, anything that could unduly affect the value returned by a function or any consequence of its execution observable in the "state of the world".

It is also possible to describe the behavior of an impure function as "non-deterministic", because it can change from one call to another unpredictably (in the sense that we cannot predict such behavior from the code of the function itself).

1.2 Advantages of pure functions. The substitution rule and equational reasoning. Composition of pure functions.

Pure functions in programming have a simple property with notable consequences. By the very definition of purity, a call to a pure function in an expression can be substituted by its value. This "substitution rule" is at the base of equational reasoning, a remarkably useful logic for reasoning about functional programs, which in particular can be used as the simplest possible mechanism of program evaluation.

Equational reasoning is applied to demonstrate the equivalence of an initial expression to a final one by going from start to end through a sequence of intermediate expressions, deriving each of them from the preceding one by applying the substitution rule, or, as is also frequently said, "substituting equals by equals".

> Equational reasoning has a wider scope. Besides being applied for evaluating functional programs, it can be used for proving their properties (in particular their correctness) or even for designing or deriving them formally from a specification. Those more advanced forms of equational reasoning, are outside the scope of this introductory article.

To illustrate evaluation by substitution, let's make the following definitions:

```
def square(n: Int): Int = n * n

def sum(a: Int, b: Int): Int = a + b

val res = square(sum(5, square(3)))
```

and apply the procedure just described to the evaluation of `res` (the == symbol can be read as "is equivalent to what follows, by the substitution principle applied for the reason indicated"):

```
square(sum(5, square(3)))
== { definition of square }
square(sum(5, 3 * 3))
== { arithmetic and definition of sum }
square(5 + 9)
== { arithmetic and definition of square }
14 * 14
== { arithmetic }
196
```

This sequence of equivalent expressions allows us to conclude that the value of `res` is 196.

Evaluation by substitution has an astounding property: the order of evaluation doesn't matter. For example, the preceding evaluation could have been made as:

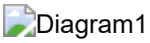
```
square(sum(5, square(3)))
== { definition of square }
sum(5, square(3)) * sum(5, square(3))
== { definition of square applied twice }
sum(5,  3 * 3) * sum(5, 3 * 3)
== { arithmetic and definition of sum applied twice }
(5 + 9) * (5 + 9)
== { arithmetic }
14 * 14
== { arithmetic }
196
```

The side-effecting code **breaks this property**, making the order of evaluation relevant and precluding in general the application of equational reasoning.

> The property of independence of the value from the order of evaluation for pure expressions, is intuitively clear and its practical implications are easy to understand. Conceptually it is rooted on an abstract theory that provides a logical foundation to functional programming: the lambda calculus. If you decide to explore that field, you will see how hard it can be to formally prove a property that we hopefully made clear just by presenting a couple of examples and appealing to your intuition.

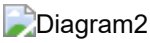
Summarizing what we have said up to this point: writing programs that use only pure functions allows us to apply the simple and powerful logic of equational reasoning to evaluate, and in general reason about, our programs.

Programming with pure functions has another great advantage. The mechanism to build programs from pure functions is also extremely simple and very powerful: function composition, the use of the value returned by a function as (one of) the argument(s) of another function. Function composition can be graphically represented using diagrams, where functions are displayed as boxes with one or more inputs (the arguments) and one output (the returned value), with the outputs of some boxes "feeding" as inputs other boxes, as in the following diagram:



The result of composing two (or more) functions is also a function, so it can be represented as a single box. In the previous example, this is hinted at by displaying a new box that encloses those belonging to the functions f11, f12, f13 and f14, and represents the function resulting from their composition (f1).

These new boxes can in turn be composed with other boxes in another diagram, and so on, without limits, "graphically" showing the boundless power of composition, as well as another of its conceptual advantages: the possibility of representing a program at different levels of abstraction. This is exemplified in the following diagram, where the functions displayed at the level of the previous diagram are represented at a higher level, in a more abstract view, by replacing the composed functions by their composition:



Equivalently, the functions composing a program at a certain level can be displayed in a more detailed view, at a lower level of abstraction, by "decomposing" some functions into the functions that compose them.

> Scala being a strong statically typed programming language, the type-compatibility of the output of a function with the input of the function it feeds to, is verified at compilation time. Here we will take that compatibility for granted, without delving at all in the intricacies of the mighty Scala type system.

Building programs by composing functions means that our programs are, in the end, just "big functions" composed by a certain number of smaller ones. Of course, we don't want those big functions to be non-deterministic. So, as **the composition of any number of functions is impure if only one of them is impure**, and as impurity implies non-determinism, we must use pure functions when applying this way of building programs.

Summarizing what we have argued in this section: if we want to maintain the advantages of equational reasoning and of program building by composition, we are restricted to program using pure functions.

1.3 The I/O monad.

As already said, there are many kinds of side effects, the most common ones being: i) input/output, and ii) all kinds of external (frequently remote) services. Do we need them? Absolutely, without them nothing observable would ever happen as a consequence of the execution of a program! Why then insist on purity? We expect that the properties of programming with pure functions discussed above can convince the reader of the advantages of not sacrificing purity to the ability of handling side effects. In Section 6 we will say something more about those advantages, after having developed a small but complete pure-functional side-effecting program.

So, how can we make the need for side-effecting code compatible with the convenience of programming using pure functions? Applying a simple and powerful (again!) trick: replace effects by their definitions as immutable data, allowing their composition as first-class values, and suspending their execution until the so-called "end of the world", that is, the moment when everything is done regarding the compositional "building" of the desired functionality and the only thing that remains is to execute the corresponding effects. Well, sure, this "end of the world" could have been called, somewhat less apocalyptically, the "end of the program" (its execution!).

> This is the third time, in this brief article, that we use the adjectives "simple" and "powerful" combined to qualify some property of pure functional programs. We believe that such a combination quite well expresses the beauty of the fundamentals of functional programming. ZIO puts this simplicity and power in the hands of a Scala programmer without resorting to the advanced features of the Scala type system or to sophisticated FP design patterns, which in the alternatives to ZIO represent strong cognitive barriers for Scala beginners. All obstacles to the development of highly concurrent applications using only the fundamentals of pure functional programming, are deliberately avoided by ZIO.

This suspension of the side effects, together with the machinery needed to compose them before their execution (which as we will soon see is based on just two operators) is what is commonly known as an IO monad. There are a handful of IO monads available for Scala. One of them, ZIO, our preferred one, was chosen in this article to illustrate the advantages of separating the abstract definition of a side-effecting program from its implementation.

Actually, ZIO is a lot more than a IO monad for Scala, as a brief look at Section 2, or even better a look at the article mentioned at the beginning of this section, should make clear.

2. ZIO as an IO monad for console input/output operations.

ZIO deals with side effects in the most general terms, but for our purposes it is necessary to consider only the effects related to input/output operations performed through the console, which ZIO provides by means of the `zio.Console` object. We will now discuss those I/O console effects within the general context of ZIO effects, which we will briefly introduce just to provide a reference frame.

The general definition of a ZIO effect (or functional effect, or just ZIO for brevity) takes the form of the `ZIO[R, E, A]` parameterized type, where the type parameters represent:

- R, the resource type i.e. the type of the resource (or resources) required to execute the effect,
- E, the error type i.e. the type of the error returned by the effect in case of a failed execution, and
- A, the result type i.e. the type of the value returned by the effect in case of a successful execution.

ZIO allows the construction of functional effects from Scala code in many ways, the most common being:

- `succeed` , a smart constructor that allows to use any value of type A to create a successful ZIO effect,

- `fail`, another smart constructor which allows to use any value of type E to create a failed ZIO effect,
- `attempt`, a function that takes as argument any segment of effectful Scala code and returns a `ZIO[Any, Throwable, A]` where A is the type of the value returned by the code segment.

A particular type of functional effects, those that do not require resources, belong to the type `IO[E, A]`, just a synonym of `ZIO[Any, E, A]`, where `Any` is the type used to represent the absence of requirements for a functional effect. The console I/O operations belong to this `IO[E, A]` type, where most frequently E is `Throwable`. For example, `Console.readLine` has type `IO[Throwable, String]` and `Console.printLine` has type `IO[Throwable, Unit]`. In the first case the value returned by the functional effect is a `String` (the line read from standard input), in the second, the useless single value of the `Unit` type.

The value returned by a functional effect can be "transformed" by applying to it a function of the proper type, via the `map` operator. `map` takes a function `f` of type `A => B` and an `IO[E, A]`, returning an `IO[E, B]`, where the value returned by the second ZIO results from applying `f` to the value returned by the first ZIO.

The informal use of the terms "transform" and "combine" with respect to ZIOs should be interpreted in the context of the processing of immutable values. Strictly speaking, for example, a transform operator does not directly transform the return value of a ZIO, but returns a second ZIO which will apply the required transformation to the value returned by the first ZIO, when the corresponding side effects are executed. In what follows we will freely use the terms "transform" (or "convert") and "combine" (or "compose") applied to functional effects, assuming that interpretation.

The value returned by a ZIO can also be fed to a function that returns another ZIO, using the `flatmap` operator. `flatmap` applies a function `f` of type `A => IO[E, B]` to an `IO[E, A]`, returning an `IO[E, B]`, where the second ZIO (the functional effect itself, not its return value as in the case of `map`) results from applying `f` to the value returned by the first ZIO.

The `map` and `flatMap` operators together form the basis of the Scala `for` comprehension, widely used as an intuitively clear representation of the sequential composition of functional effects. Said in other terms, those operators allow ZIO functional effects (in particular `zio.Console` effects) to behave as a "monad" (actually `withFilter` is also needed by a Scala for comprehension, but we can ignore it in this introduction).

> The use of for comprehensions is intuitively clear and their practical implications are well known to Scala programmers. The concept of monad has its roots in an abstract theory that gives a foundation to many important concepts in the field of functional programming: category theory. Fortunately, there's no need to learn category theory to fruitfully use monads.

A few examples may help to clarify the concepts presented so far:

```
// exampleMap
val zio1 = ZIO.succeed("This is a string")
val zio2 = zio1.map(_.toUpperCase)
val exampleMap =
  for {
    val1 <- zio1
    _ <- Console.printLine(s"zio1 succeeds with value '${val1}'")
    val2 <- zio2
    _ <- Console.printLine(s"zio2 succeeds with value '${val2}'")
  } yield ()

// exampleEcho
```

```
    Console.readLine("Echo ... please type something\n").
      flatMap(line => Console.printLine(line).
      map(_ =>"You typed '" + line + "'"))

  // exampleEcho2
  val exampleEcho2 =
    for {
      line <- Console.readLine("Echo ... please type something\n")
      _ <- Console.printLine(line)
    } yield ("You typed '" + line + "'")
```

The first ZIO example, `exampleMap`, shows how `map` can transform the result value of a functional effect. The success values of zio1 and zio2 are displayed using `Console`, an already mentioned service that provides functions to perform simple I/O operations on the input/output channels of a standard console application.

The visible results at the console of running `exampleMap` are:

```
zio1 succeeds with value 'This is a string'
zio2 succeeds with value 'THIS IS A STRING'
```

The second ZIO example, `exampleEcho`, shows how `flatMap` can be used to communicate two ZIOs executed in sequence.

The third ZIO example, `exampleEcho2`, is totally equivalent to the second, but it is written using a for comprehension. Comparing these two examples we can clearly see how:

- the use of `flatMap` in the second example is syntactically sugared in the third by means of the `<-` operator (which allows "retrieving" the result of a functional effect to "feed" it to another one within the comprehension), and
- the use of `map` in the second example is syntactically sugared in the third by means of the `yield` operator (which allows defining the result of the entire comprehension).

Note also how `exampleEcho2`, thanks to the `for` syntax, naturally reads almost like an equivalent imperative program. This increases the intuitive appeal of `<-` and `yield`, that certainly `flatMap` and `map` don't provide.

ZIO applications can be run as normal Scala apps by embedding them within an object extending `ZIOAppDefault` and overriding the `run` function:

```
import zio._

object ZIOExampleApp extends ZIOAppDefault {

  def fnReadLine(prompt: String) =
    for {
      _ <- Console.printLine(prompt)
      line <- Console.readLine
    } yield line

  val appReadLine =
    for {
      line <- fnReadLine("Please type a line")
      _ <- Console.printLine(s"You typed: '${line}'")
    } yield ()
```

```
  override def run = appReadLine
}
```

Here, the function `fnReadLine` receives a `String` and returns an IO with result type `String`. This IO is called in `appReadLine` within a `for` comprehension, and its result is used in another line of the same comprehension. `appReadLine` is then used as the `run` function of a ZIO app (one that extends `ZIOAppDefault` as said before).

To finish our brief presentation of ZIO operators, we now discuss some important combinators for functional effects:

- `orElse`, allows to execute an alternative functional effect when, and if, the main one fails. The value returned by the composed effect is that of the main effect, if it succeeds, and that of the alternative one, if it fails.
- `zip`, compose two functional effects by executing both and returning the 2-tuple of their returned values. If one of the two effects fails, the entire composed effect fails.
- `zipRight`, the same as `zip`, but returns only the value returned by the second effect.
- `zipLeft`, again the same as `zip`, but returns instead only the value returned by the first effect.

Here we have a couple of examples using the combinators just presented:

```
import zio._

val helloWorld = ZIO.succeed(print("Hello ")).
                 zipRight(ZIO.attempt(print("World!")))
val helloWorld2 = Console.print("Hello ").
                  zipRight(Console.print("World!"))

val primaryOrBackupData = ZIO.readFile("primary.data").
                          orElse(ZIO.readFile("backup.data"))
```

The first two examples print "Hello World!" to the console as the result of executing two ZIOs in sequence, discarding the value of the first one and returning the value of the second (both equal to the useless value of type `Unit`, anyway). The difference between those examples lies in the fact that the first uses `succeed` and `attempt` to transform Scala side-effecting code into functional effects, while the second uses directly functional effects provided by `zio.Console`.

The third example shows a typical use of `orElse` to provide a fallback to an IO operation that could fail.

3. The abstract definition of a yes/no interactive dialogue and its implementation as a console application.

To further elaborate on the main ideas presented so far, we will apply them to the implementation of a simple kind of interactive dialogues, those that allow only responses of the "yes/no" type, which can be represented by the `Dialogue` data type defined in the code that follows, which also presents an example instance of that type:

```
sealed trait Dialogue

case class Ask(question: String, yesContinuation: Dialogue, noContinuation: Dialogue) extends Dialogue
```

```
case class Stop(conclusion: String) extends Dialogue

val exampleDialogue =
  Ask("Do you know ZIO?",
    Ask("Do you like it?",
      Stop("Good!"),
      Stop("I can't believe it!")),
    Stop("What a pitty!"))
```

This abstract definition of yes/no dialogues can be seen as a specification of a program that implements that kind of dialogues. We can also say that an implementation plays the role of an "interpreter" of the abstract definition. In this section we will develop a console interpreter for these yes/no dialogues, using the services provided by `zio.Console`.

Applying the for comprehension provided by ZIO for functional effects, the abstract recursive `Dialogue` definition almost literally translates into a recursive console implementation that simply pattern-matches over the two case classes that extend the `Dialogue` trait:

```
def consoleDialogue(dialogue: Dialogue) = dialogue match {
  case Ask(question: String,
           yesContinuation: Dialogue, noContinuation: Dialogue) =>
    for {
      bool <- askBooleanQuestion(question)
      _    <- if (bool) consoleDialogue(yesContinuation)
              else consoleDialogue(noContinuation)
    } yield ()
  case Stop(conclusion: String) =>
    Console.printLine(conclusion)
}
```

Here, we assume the availability of `askBooleanQuestion`, a functional effect whose obvious role would be to ask by console a yes/no response to a given question, returning a `Boolean` representation of the response (retrying if the console response is not equal to one of the allowed options, until it is).

In other words, the expected functionality of `askBooleanQuestion` consists of: i) print to the console the question, ii) enter a cycle of reading an answer from the console until a valid one is typed by the user (we will take as valid answers the strings "y" and "n" with the obvious interpretations), and iii) return the valid answer interpreted as a `Boolean`.

If we assume this time the availability of a function `getBool` that implements the cycle to ask for a "y"/"n" response, `askBooleanQuestion` can be implemented as:

```
def askBooleanQuestion(question: String) = for {
  _    <- Console.printLine(question)
  bool <- getBool
} yield bool
```

Finally, `getBool` can be implemented as:

```
def getBool(): IO[IOException, Boolean] =
  for {
    input <- Console.readLine
```

```
      bool  <- ZIO.fromOption(makeBool(input)) orElse
               (Console.printLine("Please type 'y' or 'n'") zipRight getBool)
    } yield bool

  def makeBool(s: String): Option[Boolean] = {
    if (s == "y") Some(true)
    else if (s == "n") Some(false)
    else None
  }
```

`makeBool` is a pure function that converts a console `String` answer to an `Option[Boolean]`. It returns `Option[Boolean]` instead of `Boolean` to consider the possibility of an incorrect console answer (which is converted to `None`).

`getBool` is a functional effect that implements the cycle to:

- read a line from the console,
- convert it to an `Option[Boolean]`, and
- re-enter the cycle through a recursive call if the conversion fails, or yield the corresponding `Boolean` if the conversion succeeds.

`getBool` works using `ZIO.fromOption` to convert the `Option[Boolean]` value returned by `makeBool` to a successful/failed ZIO, which is composed, using `orElse`, with the following alternative ZIO (that is executed only when the left argument of `orElse` is a failed ZIO because `makeBool` returned `None`):

```
(Console.printLine("Please type 'y' or 'n'") zipRight getBool)
```

This ZIO is obtained by the sequential composition, using `zipRight`, of the display of a prompt message with `getBool` itself, thus implementing the desired recursive calling of `getBool` until a valid answer is obtained from the console.

ZIO features stack-safety for arbitrary recursive effects. The free use of recursion allows us to recover an important basic technique used in functional programming to define arbitrary control-flow structures, usually excluded outside quite constrained bounds by the threat of stack overflow.

> This safe-recursion feature can be seen as one more of the many ways in which ZIO facilitates getting back to the fundamentals of FP to solve complex problems in a simple manner.

4. Advantages of an abstract definition of yes/no dialogues.

So, now we have an interpreter for our yes/no dialogues (the function `consoleDialogue` explained in the previous section). A sample of its execution could be:

```
Do you know ZIO?
y
Do you like it?
x
Please type 'y' or 'n'
y
```

```
Good!
```

Nice, but let's say that we decide to enhance our dialogs by first greeting the users that answer them.

We could be tempted to do that directly in the interpreter by asking the user's name with `readLine` and then greeting the user with `printLine` and continuing with the dialogue. But, wouldn't it be better to continue the dialog asking first if the user wants to? That, and other possible improvements, seem to ask for an intervention at the "model" level of our program, to use a common distinction with the "view" level at which the interpreter of our abstract definition of dialogues works. And, yes, that is the best way.

In fact, it is quite simple, and completely general (that is, without any consideration regarding specific aspects of a given implementation), to define a function on a dialogue that, given a name, returns a "greeting dialogue" version of it. The best place for that function should be, naturally, the trait defining the dialogues themselves:

```
sealed trait Dialogue {
  def greetFirst(name: String): Dialogue =
    Ask(s"Welcome $name, are you ready to continue?", this, Stop(s"See you later ${name}."))
}
```

Now we have a function that actually defines a new kind of dialogues at an abstract level. The interpreter necessary to implement these new dialogues is trivial, because it can make good use of the previous one:

```
def greetFirstConsoleDialogue(dialogue: Dialogue): IO[Exception, Unit] =
  for {
    name <- Console.readLine("What is your name?\n")
    _    <- consoleDialogue(dialogue.greetFirst(name))
  } yield ()
```

These definitions clearly show that the change we made to the `Dialogue` definition, preserves a neat distinction between the abstract definition of the dialogues and their implementation, which allows us to work independently (and as a consequence more easily) at the "model" and the "view" levels of our program.

> The model-view separation can be seen as a "first-level" application of the separation of concerns design principle, which aims to develop modular software systems. This principle can be superbly applied in ZIO using `Zlayer`s, which are software layers used to define the modular structure of an application. A complete example of a ZIO application developed using these layers can be found in the article mentioned at the beginning of Section 1.

Here we have a sample of the execution of the new interpreter:

```
What is your name?
Sam
Welcome Sam, are you ready to continue?
y
Do you know ZIO?
x
Please type 'y' or 'n'
y
Do you like it?
```

```
n
I can't believe it!
```

We could define other operations on dialogues, for example one (let's name it `batchDialogue`) that, given a `String` representing a sequence of "y"/"n" responses, returns the final "result" of a dialogue (the `conclusion` value of the last `Stop` visited) as an `Option[String]` (`None` should be used as result for a sequence of characters that is not valid for the given dialogue).

Then, we could demonstrate some properties of our operations, for example, necessary conditions for the input of `batchDialogue` to be acceptable (i.e. to return a `Some`).

We could also implement new algorithms and prove the correctness of their implementations, for example implement `batchDialogue` as a restricted form of traversal of the tree representing a dialogue.

Or prove some rules that characterize the relationship between the defined operations, like for example (kind of trivial, but anyway):

```
if d.batchDialogue(s1) = s2 then, for all s3, d.greetFirst(s3).batchDialogue("y" ++ s1) = s2
```

Our dialogues being essentially binary trees, some properties of those data structures could be "translated" to properties of our `Dialogue` data abstraction. All those general properties, as well as the more specific properties of the operations we defined, would allow us to reason about yes/no dialogues in an essential way, valid not only for some specific implementation but also for any other that correctly implements the abstract definition.

Because, yes, we could not only write in many ways a `ZIO.Console` implementation of our dialogues, but also develop a web implementation or an implementation for Android devices, or whatever, taking always as the base for a new implementation the abstract definition of the dialogues and of the operations defined on them.

All those implementations, if correct, would be in essence equivalent. That is, in practical terms, diverse ways of doing essentially the same thing.

In the end we would finish having a data structure with operations defined on it and with rules (or "laws") that govern their application and help to reason formally about them. Does this sound familiar as a concept? You guessed it, we would finish with an "algebra" of yes/no dialogues. That's amazing, considering that we are not mathematicians but (functional) programmers!

5. Concluding remarks. References.

An I/O monad allows programming with side effects using only pure functions, thus combining the advantages of pure functional programming with the inevitable need for effects through which our programs can interact with the world outside of them.

Those much-touted advantages of pure functional programming are: i) constructing programs using function composition, and ii) thinking about programs applying equational reasoning.

i) Regarding composition as an effective tool for building programs, we hope that the reader found our simple "semi-graphical" arguing in Section 1 persuasive (well, if you were not persuaded beforehand). Actually, we don't need to sing the praises of compositionality using diagrams. As software developers, we know very well that

solving a programming problem by first dividing it into simpler parts and then putting together the solutions of the parts as a solution of the whole,

is the "divide-and-conquer" strategy that allows us to build correct software no matter the degree of complexity involved. The crux here is that this strategy is applied in the simplest and most "natural" way conceivable by the composition of pure functions (there are other ways, of course, because software building, even if not using a pure functional approach, simply cannot do without this strategy).

ii) Regarding equational reasoning as an effective tool for thinking about programs, we must admit that probably as ZIO programmers we will never use it explicitly (that could remain an academic exercise for Haskell programmers). But anyway, as a consequence of using pure functions, the mental models of our programs will be a lot more tractable than that of the equivalent imperative ones, helping us to correctly reason about the programs we write, to understand them, compose them, transform them, etc.

> This is again an informal statement appealing to your intuition. If you want more "formal evidence" supporting it, just take a look at how difficult it can be to prove the correctness of an imperative program (even a very simple one) using, for example, Hoare logic. Then, compare that with the straightforward - we would even say natural - way in which the correctness of a pure functional program can be proved applying a logic based on equational reasoning (you can find beautiful examples in the book on Haskell cited in the references).

ZIO implements an I/O monad for Scala, but ZIO is a lot more than that. In fact, ZIO can be defined as **an ecosystem for building purely-functional, highly concurrent, asynchronous, efficient, back-end applications in Scala** , but the aspects of ZIO beyond its use as a console I/O monad are clearly outside our current scope.

> As you may have noticed, there are deep theoretical foundations behind the concepts that we informally discussed. Fortunately, we don't need a PhD in computer science to take great advantage of the fundamental concepts of functional programming as ZIO software developers. However, knowing that we are backed by mathematically sound theories is somewhat reassuring, isn't it?

Finally, we provide some references that may be useful to delve deeper into the concepts previously presented.

The `Dialogue` example was taken from a talk by Andres Loeh about the IO monad in Haskell, which clearly explains concepts that are relevant not only to Haskell but to any functional programming language.

In Haskell you cannot write impure functions: an IO monad exists by default, and there is no way to escape from it. If you are interested in the roots of many of the ideas presented here, even a brief foray into Haskell might well be worth it. For a deeper incursion we recommend this excellent book: Graham Hutton - Programming in Haskell. On the back cover you can read a pertinent comment: "The skills you acquire by studying this book will make you a much better programmer no matter what language you use to actually program in (Erik Meijer)".

To learn more about ZIO, you can find many useful guides and good reference material on the official site.

Another site with interesting material is that of Scalac, where you can find, among others, ZIO console apps larger than ours, which you can face more motivated and prepared after reading this introduction.

The "bible" about ZIO is Zionomicon, whose main author is John A. De Goes, the creator of ZIO. In the first chapter there is a history of ZIO that mentions its birth as an IO Monad for Scala with strong emphasis, from the very beginning,

on asynchronous and concurrent programming.

The Scala code of this article, written using Scala 2.13, can be found here.

A Scala 3 version is available here.