About the Direct Syntax for Monadic Effect Systems

Ruslan Shevchenko [1]

[1] *The Institute of Software Systems of the National Academy of Sciences of Ukraine, Akademika Glushkova Avenue, 40, building 5, Kyiv, 03187, Ukraine*

Abstract

The article describes the ergonomic programming language interface for working with monadic effects, which encapsulate the logic of computation and associated non-computational operations. It describes representations of computation in the direct form with the direct context encoding technique in the Scala language. A Scala compiler plugin, available as an open-source, translates direct representation into the monadic form. Also discussed is conditional effects compilation to organize cross-platform interfaces that combine different methods of implementing effects on different platforms.

Keywords

effects, monads, Scala, programming language, and program transformation

1. Introduction

Effect systems have taken an essential place in modern programming. In the community of developers, we can see two views on the effects:

1. An effect is a program property that changes the execution environment. Unlike pure computations, effects are not reflected in the language's type system. This view is characteristic mainly of developers in imperative programming languages.
2. An effect is a property of a program's interpretation that changes the interpretation process. Effects are reflected in the type system as higher-order types, a view characteristic of functional programming.

In industrial programming, the first view wins regarding development speed and fewer concepts to master. However, the cost is the inability to automatically analyze effects using the type system and more incredible difficulty in finding errors. Eventually, these concepts still appear as exception-free code descriptions and analysis tools. However, programmers will use them later in the development stage. Such an approach speeds up the initial development cycle. The functional approach typically represents the type of an expression with effects as a monadic expression F[T], where T is the type of expression without explicit effects in an imperative programming language.

We will use Scala as our representation language. Standard monad signature includes two operations, *flatMap* and *pure*:

```
def flatMap[A,B](fa: F[A])(f: A => F[B]): F[B]

def pure[A](a: A): F[A]
```

Analogical signatures in the Haskell standard library bind for *flatMap and return for pure.*

[1] *14th International Scientific and Practical Conference from Programming UkrPROG'2024, May 14-15, 2024, Kyiv, Ukraine*

* Corresponding author.

ruslan@shevchemko.kiev.ua (R. Shevchenko)

0000-0002-1554-2019(R. Shevchenko)

For a detailed description of monadic-type interfaces and effects applications, look at Moody[1] or Wadler[1,2].
Classic examples of monadic effects are input/output operations, usually represented as IO[X], or exceptions, represented as Try[X].
An interpreter of an effect monad is a function that transforms a monadic expression into its value and performs the side effects. We can describe the type of interpreter function in meta-notation that reflects the computation environment.
Now, let's move from monads with single effects to effect systems. Suppose we are developing a program that performs both input and output and can terminate unexpectedly. What type can it have? It is on IO[T] and not Try[T]. It is some kind of monad that includes both effects. There are many ways to build such monads; let's present the most well-known ones:

One practical approach, often used in practice, is the 'god monad.' This monad combines input/output processing and error generation, making it a convenient choice when the number of effects used is small. While it may not be a full-fledged effect system, it provides a practical solution. However, unlike full-fledged effect systems, it is only possible to add a new effect or replace the implementation of an existing one by rewriting the monad interpreter. On the other hand, we can say that any actual semantics of the executable language assume that some 'god monad' is defined by the computation environment.

Historically, the first full-fledged effect stack was built based on monad transformers. The main idea is quite simple – the logical type for the result of a program that has input/output and can generate exceptions is IO[Try[T]]. We cannot use *IO[T]* directly, but we can rewrite *IO* as *IOT[F[], T]*. The interpreter of this monad converts *IOT[F,T]* to *T* using the interpreter of *F*. (Or, in some cases, you can build a partial interpreter, which can already be called an effect handler, that converts *IO[F, T]* to *F[T]*. Then, having the interpreter of F[T], we can build a full interpreter of the complex monad.) This scheme is simple and intuitive. This approach is used in industrial functional programming when we have one or two effects, which is quite a standard case. However, when working with an extensive list of effects, performance issues arise, as each operation has to go through a series of calls for each monad in the stack.

The next step is to use horizontal composition instead of nesting, i.e., separating the "monad-containing effects" from the pure effects. For example, let us have effect types $(E_1 \cdots E_n)$, and the programming language capabilities allow us to construct a type-level list $L = (E_1 \cdots E_n)$. Then, we can build a monad $Eff[X] = F[L, X]$ with effect interpreters of the form: $I[E_k]: F[L,X] \rightarrow F[L - E_k]$, where $L - E_k = (E_1 \cdots E_{k-1}, E_{k+1}, \cdots E_n)$ the list of effect types L is without E_k. The effect types E_i themselves do not necessarily have to be monads.

The Freer monad[6] is a classic example of such a construction. Currently, in the context of industrial programming, when talking about a monadic effect system, it generally refers to a similar construction with a series of optimizations. For Haskell, there are about ten implementations, and for Scala, several systems are currently in development, such as eff [7], kryo [8], turbolift [9].

Another approach for effect description is the so-called tagless final style of description [11], where effects are referred to not as specific types but as type classes that implement some API [10]. Indeed, the only thing that characterizes an effect is a set of methods. For example, a hypothetical input/output effect can be characterized by a pair of functions:

```
trait StdIO[X] {
 def write(line: String): StdIO[X]
 def read: StdIO[String]
}
```

If we can "push" these functions into the main monad, then this API can be written as:

```
trait StdIO[F[_]] {
   def write(line: String): F[Unit]
   def read: F[String]
```

```
}
```

Then, use this type class as a characteristic of the effect monad, regardless of how it is constructed—as a standalone effect monad, an element of a monad transformer, or an element of an effect system.

Now we can formulate the problem that direct representation of monadic effects solves: we have two methods of organizing effects. Can we unify these two forms in such a way that it is possible to use monadic effects without complicating the syntax, while still allowing the analysis of expressions with effects using type system?

This problem has two practical applications. The first is to simplify working with monadic interfaces for developers, removing syntax overhead of the monadic interfaces, and the second is for building multi-platform systems with different runtime capabilities on different platforms. Then, effects that can be represented in direct form on one platform (for example, asynchrony on post-Loom JVM) should be monadic on a platform without support for continuations, such as JavaScript.

2. Direct Syntax

Methods for easing syntactic burden appeared simultaneously with monadic interfaces. The approaches can be divided into two groups:

- Special constructions that create a 'sub-language' of the programming language for monadic DSLs.
- Transform regular language constructs into monadic form inside the scope of established pseudo-operators.

An example of the first approach is do-notation in Haskell [11], Computation Expressions in C# [12], and for-comprehensions in Scala [13]. On the one hand, this is convenient in implementation because the compiler can transform these syntactic constructs before type analysis. On the other hand, the user sees two similar but different languages, which causes additional difficulties in learning and application. Below is how typical request processing logic looks in Scala monadic DSL:

```
def myFun: IO[HttpReply] = {
 for {result1 <- talkToServer("request1", None)
      _ <- IO.sleep(100.millis)
      results2 <- talkToServer("request2", Some(results1.data))
      _ <- if results2.isOk then
             for { _ <- writeToFile(results2.data)
                   _ <- IO.println("done")
             } yield ()
           else
             IO.println("abort")
      } yield results2
}
```

An example of the second approach is bind notation in Idris [14] and async/await (though limited to asynchrony) in C# and most modern programming languages [15]. The author has developed a similar system for Scala: dotty-cps-async 16][17], which supports a pair of pseudo-operators reify/reflect (or async/await as traditional synonyms) for any monad as macros. Practically, this solution partially addresses the issue of reducing cognitive load when using asynchronous APIs and "softening" the learning curve. Here is the same code fragment as in the previous example, rewritten with the help of the `dotty-cps-async`:

```
def myFun: IO[HttpReply] = async[IO] {
 val results1 = await(talkToServer("request1", None))
 await(IO.sleep(100.millis))
 val results2 = await(talkToServer("request2", Some(results1.data)))
```

```
 if results2.isOk then
   await(writeToFile(results2.data))
   await(IO.println("done"))
 else
   await(IO.println("abort"))
 results2
}
```

But we can say that syntax overhead is fully eliminated: issues that are still left are:

- Since most operations are monadic in effect systems, the source code is still overloaded with reflect (or await) operators that do not carry non-technical meaning. So, in the example above, see an *await* expression in almost every line.
- The second problem is specific to effect systems. In the example above, we work with a single IO monad, which we specify as the type parameter async[IO]. If we work with effect systems, the monad type is complex, and the async expression will look like:

  async[[X] => Eff[IO::*Error::*Config, X]]

  The programmer has to write this full-type signature, which sometimes changes depending on the effects used. Writing big signatures when the compiler can deduce them from the code is an ergonomic issue.

To address the issue of the overhead of awaits inside the async block, the early versions of dotty-cps-async proposed an automatic coloring mode based on implicit conversions. We allow implicit conversion between IO[T] and T if the only use of the asynchronous expression is to take or ignore the value. We report an error if our expression, which we apply unwrapping conversion, is also used differently (for example, passed to another function that accepts IO[T] instead). An example of using automatic coloring with the same functionality as in the previous code fragment is:

```
def myFun: IO[HttpReply] = async[IO] {
 val results1 = talkToServer("request1", None)
 IO.sleep(100.millis)
 val results2 = await(talkToServer("request2", Some(results1.data)))
 if results2.isOk then
   writeToFile(results2.data)
   IO.println("done")
 else
   IO.println("abort")
 results2
}
```

As we can see, there is some visual improvement, but this solution did not gain widespread adoption because analyzing such code is somewhat more complex (for instance, why is there no await near result1, but there is one near result2?). To independently understand which types are inferred, one needs to consider not only the language rules but also how each async expression is used.
Despite the macro performing type analysis and generating an error for potentially incorrect usage, such transformations were perceived by developers as unsafe.

3. Using context functions for a direct style

Instead of the automatic coloring mode, a compiler plugin has been developed that allows the direct style of monadic expressions using contextual arguments and functions.

Let's remember what are context arguments and context functions in Scala3:

A contextual argument called an implicit argument in Scala 2, is a function argument that the compiler can automatically synthesize on the call side without mentioning this argument in the program code. Argument lists of context methods and functions are marked with the `*using*` keyword,

and the compiler, when generating the function call, will look for the values of these parameters in the context. For example, the following definition:

```
 def write[A](a: A)(using Writer[A]): Unit
```

defines a function with a contextual parameter *Writer[A]*, and we can call it as *write(10)*.

Context functions (*A ?=> B*) represent computations that depend on context A and return B. We can understand it as a shortcut for the lambda function that accepts the implicit argument. A valuable feature for us is the short syntax of context lambda functions: in a method that takes a context function A ?=> B as input, you can pass an expression of type B. Inside this expression, a programmer can use the context value of A (notation: *summon[A]*) and, therefore - other context values that depend on *summon[A]*. For example, let's say we have an object from which we can derive a Writer for elementary types:

```
trait Writers {
 given Writer[Int] ...
 given Writer[String] ...
}
```

and a function

```
  def withJsonWriters[A](f: Writes ?=> A): A
```

Then, the call to such a function can take the form:

```
withJsonWriters {
 write(1)
 write("abc")
}
```

Returning to monadic effects: we can represent a monadic computation F[X] as a context function that returns X: CpsDirect[F] ?=> X. Using the properties of context functions described above, we can freely use the results of asynchronous calls within the scope of CpsDirect[F] as values of type X.

The compiler plugin transforms functions and methods that contain the context parameter CpsDirect[F] and returns values of type X into methods that return F[X] and converts the body of these methods into monadic form. The previous example is now shown:

```
def myFun(using CpsDirect[IO]): HttpReply = {
 val results1 = talkToServer("request1", None)
 sleep(100.millis)
 val results2 = talkToServer("request2", Some(results1.data))
 if results2.isOk then
    writeToFile(results2.data)
    println("done")
 else
    println("abort")
 results2
}
```

(Here, we implicitly assume that the API is also converted into a direct context representation.)

We can call functions with the *CpsDirect[F]* parameter from async blocks and other direct context functions and freely use the pseudo-operator *await/reflect* to convert *F[T]* to *T* when necessary.

The transformation schema is simple: we transform a function definition to return a value wrapped in monad if any of the arguments (regardless of whether implicit or explicit) is CpsDirect[F].

$$\frac{C_F[|def\ f(a^i_j : A^i_j) : B|]}{def\ f(a^i_j : \grave{A}^i_j) : C_F[|B|]}.\ f \exists A^i_j = CpsDirect[F]$$

where

$$\grave{A}^i_j = \begin{cases} CpsMonadContext[F] & A^i_j = CpsDirect[F] \\ A^i_j & otherwise \end{cases}$$

Let's name functions that satisfy this condition: a context-direct function, $f \in Direct$.

On the call side, we should add to cps-transformation schemas, defined in[16] appropriate translations rules:

- for direct context functions without lambda arguments.

$$\frac{C_F[|f(a^j_i)|]}{F.\ flatMap_{i,j}([C_F|a^i_j|])(\hat{b^i_j} \to f(b^j_i))}.\ f \in Direct[F] \wedge a^i_j\ not\ a functional$$

- for high-order direct context function with nontrivial lambda arguments we can substitute async shifted version in the same way as for ordinary functions.

$$\frac{C_F[|f(a^j_i)|]}{F.\ flatMap_{i,j}([C_F|a^i_j|])(\hat{b^i_j} \to \grave{f}(b^j_i))}.\ f \in Direct[F] \wedge \exists \grave{f} = async\ shifted\ f,\ a^j_i \in \Lambda$$

Here $f(a^i{}_j)$ is a shortcut for a function call with possibly more than one argument list: $f(a^1{}_1, \cdots a^1{}_{n_1}) \cdots (a^k{}_1 \cdots a^k{}_{n_k})$
A full set of CPS transformation rules is presented in Appendix A.

Also, to ensure the correctness of transformations, we impose a number of restrictions on expressions of type CpsDirect[F] - they cannot be used in assignment statements, function return values, or matched with type parameters. In the future, these restrictions may be expressed using capture tracking, which is currently being developed as an experimental extension of Scala.

We can call the direct context function from the async blocks, where our API assumes the existence of CpsMonadContext[F]. CpsDirect[F] is implemented as an opaque type over CpsMonadContext[F], where conversion is performed by the compiler plugin.

The resulting model of asynchrony is quite close to the model of suspended functions in Kotlin. Note that, unlike the "automatic coloring" mode of functions, here the types F[X] and CpsDirect[F] ?=> X are different types for the programmer, and for each function, we need to choose how it should be called. Thus, the problem of coloring has not disappeared: it just became possible to lower the syntax noise by changing the main method of invocation to quasi-synchronous.

Note that using only direct syntax notation limits the expressivity of the language. We need to mix monadic and direct syntax to express explicit parallelism. For instance, suppose we need to read several data streams in parallel. If we assume that fetch is also written in direct form, then the following block of code

```
def readFirstN(urls: Seq[String])(using CpsDirect[Future]): Seq[String =
 urls.map(url => fetch(url))
```

will read all the data sequentially, one after another. If we need parallel processing, we have to convert the direct API into monadic form:

```
def readFirstN(urls: Seq[String])(using CpsDirect[Future]): Seq[String] =
 urls.map(url =>
   asynchronized(fetch(url))
 ).map{
   future => await(future)
 }
```

Therefore, for direct syntax, a pseudo-operator `asynchronized` (or `reified`) is introduced, which converts a pseudo-synchronous expression into a monadic one, dual to `await`. The translation function just leave the underlaying expression unchanged, assuming it already has wrapped type.

$$\frac{C_F[|asynchronized(a)|]}{C_F[|a|]}$$

In the case where F[_] is a complex monad of effects, a concise syntax can be developed to specify the set of effects in F as a set of contexts or type constraints of F. The direct syntax can contain not a context for the monad but the context for effects and the compiler plugin can assemble the needed monad from the set of effects. Yet one option is to abstract from the specific effect system F in the form of a tagless final. The optimal form of such a record is an open question that is the subject of further research.

4. Compatibility with the algebraic effect systems.

Another common development problem is supporting cross-platform libraries where the runtime environment has different properties on different platforms. Specifically, on the Java platform starting from version 21, it has become possible to develop algebraic effect systems based on continuations, while a similar API in non-monadic form is impossible in scala-js. Is it possible to build a compilation mode so that the same code is compatible with an asynchronous programming library on the JVM based on virtual threads and a system based on the asynchronous concurrency model of JavaScript?

The main problem is that non-monadic effects are currently not reflected in the type system, so we do not have the information for correct translation from the side of the continuation-based system. Therefore, we cannot transfer code from a continuation-based system to a monadic one. But we can do the opposite: in a monadic system, the types have all the necessary information. Thus, we can write the API of the continuation system as a direct representation of the monadic form, then compile it into JavaScript as a monadic form, and in JVM, simply erase the context parameter (i.e., the plugin should convert CpsDirect[F] ?=> X to F[X] on JavaScript, and to X on JVM for effects that are implemented through JVM continuations). This will allow programs to use the same basic API on both platforms.

In gears[20], suspended functions always accept context parameter of the special type `Async` . Therefore, it is possible to implement source-compatible API-s in direct form, where define Async as a context type in our monadic representation.

5. Conclusions

Direct context representation of effects can be a useful tool in software development as it reduces the cognitive load on the programmer from syntactic noise while preserving information about effects in the types. Additionally, using such a representation, it is possible to build cross-platform applications that can use both monadic and non-monadic effect systems.

In the dotty-cps-async project, a Scala compiler plugin has been developed that converts direct context representation into monadic form. Further research directions include finding the optimal form for supporting complex effect systems, continuing experiments in the direction of cross-platform effect translation, and improving the generation of the monadic representation of the program.

6. Appendix A: Rules for Cps Transformation

Here is a full combined list of rules for monadic cps transform of in scala in dotty-cps-async without optimization.
Here
Λ - Set of lambda functions.
Direct - set of direct functions, which accept CpsDirect[F] parameter
I_Λ - set of invariant lambda functions, i.e. functions like $X \Rightarrow F[Y]$ or $X \Rightarrow y: y \in I_\Lambda$

name	rule
trivia	$\frac{C_F[\|t\|]}{F.pure(t)}$
t is an expression without async subexpressions.	
sequential	$\frac{C_F[\|\{a;b\}\|]}{F.flatMap(C_f[\|a\|])(\_ \Rightarrow C_F[\|b\|])}$
valdef	$\frac{C_F[\|\{val\ a = b; c\}\|]}{F.flatMap(C_f[\|b\|])(a \Rightarrow C_F[\|b\|])}$
condition	$\frac{C_F[\|if(a)then\ b\ else\ c\|]}{F.flatMap(C_f[\|a\|])(a \Rightarrow if(a)\ then\ C_F[\|b\|]\ elseC_f[\|c\|])}$
match	$\frac{C_F[\|c\ match\ case\ (r_i \Rightarrow v_i)\|]}{F.flatMap(C_f[\|c\|])(c \Rightarrow match\ c\ case\ r_i \Rightarrow C_F[\|v_i\|])}$
while	$\frac{C_F[\|while(a)b\|]}{whileHelper(C_F[\|a\|], C_F[\|b\|])}$

```
def whileHelper[F:CpsMonad](c:F[Boolean])(body: F[Unit]): F[Unit] =
 F.flatMap(c) { b =>
  if b then
    F.flatMap(body) { _ => whileHelper(c)(body) }
  else
    F.pure(())
 }
```

try/catch/finally	$$\frac{C_F[\lvert try(a)\ catch(e_i \Rightarrow c_i)\ finally(d)\rvert]}{F.flatMap(\ F.flatMapTry(C_F[\lvert a\rvert])\begin{pmatrix}case\ Failure(e_i) \Rightarrow C_F[\lvert c_i\rvert] \\ case\ Success(x) \Rightarrow F.pure(x)\end{pmatrix},\ r \Rightarrow C_F[\lvert d\rvert].map(\_ \Rightarrow r)\ )}$$
throw	$$\frac{C_F[\lvert throw\ e\rvert]}{F.error(e)}$$
lambda	$$\frac{C_F[\lvert a_i \Rightarrow b\rvert]}{a_i \Rightarrow C_F[\lvert b\rvert]}$$
application (non-ho, non-direct)	$$\frac{C_F[\lvert f(a_j^i)\rvert]}{F.map(C_F[\lvert a_i\rvert])(f)} a_j^i \notin \Lambda \vee a_j^i \in I_\Lambda, f \notin Direct$$
application (non-ho, lambda)	$$\frac{C_F[\lvert f(a_j^i)\rvert]}{F.flatMap(C_F[\lvert a_i\rvert])(b_j^i \Rightarrow C_F[\lvert f\rvert](b_j^i))} a_j^i \notin \Lambda \vee a_j^i \in I_\Lambda, f \in \Lambda$$
application (non-ho, direct)	$$\frac{C_F[\lvert f(a_j^i)\rvert]}{F.flatMap(C_F[\lvert a_i\rvert])(f)} a_j^i \notin \Lambda \vee a_j^i \in I_\Lambda, f \in Direct$$
application (ho)	$$\frac{C_F[\lvert f(a_j^i)\rvert]}{F.flatMap(C_F[\lvert a_i\rvert])(\grave{f})} a_j^i \in \Lambda \wedge a_j^i \notin I_\Lambda$$
$\exists \grave{f} = async\ shifted\ f$	
asynchronized	$$\frac{C_F[\lvert asynchronized(a)\rvert]}{C_F[\lvert a\rvert]}$$
await (non-conversion)	$$\frac{C_F[\lvert await_G(a)\rvert]}{Contex[F].adopt(a)} F = G$$
await (conversion)	$$\frac{C_F[\lvert await_G(a)\rvert]}{Contex[F].adopt(CpsMonadConversion_{[G,F]}(a))} F \neq G$$

References

[1] Eugenio Moggi, Notions of computation and monads, Information and Computation, 93 1 (1991) [doi:10.1016/0890-5401(91)90052-4, pdf]

[2] Philip Wadler. 1995. Monads for Functional Programming. In Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Springer-Verlag, Berlin, Heidelberg, 24–52.

[3] Philip Wadler. The essence of functional programming. In Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '92). 1992 Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

[4] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '95). 1995. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[5] Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and modular algebraic effects: what binds them together. In Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019). 2019. Association for Computing Machinery, New York, NY, USA, 98–113. https://doi.org/10.1145/3331545.334259

[6] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. SIGPLAN Not. 50, 12 (December 2015), 2015, 94–105. https://doi.org/10.1145/2887747.2804319

[7] eff: [cited 08.04.2024] https://github.com/atnos-org/eff

[8] kyo: [cited 08.04.2024] https://github.com/getkyo/kyo

[9] turbolift [cited 08.04.2024] https://marcinzh.github.io/turbolift/

[10] CARETTE, J., KISELYOV, O. and SHAN, C.-C. (2009) 'Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages', Journal of Functional Programming, 19(5), pp. 509–543. doi:10.1017/S0956796809007205.

[11] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. . A history of Haskell: being lazy with class. In Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III). Association for Computing Machinery, New York, NY, USA, 2007, 12–1–12–55. https://doi.org/10.1145/1238844.1238856

[12] Petricek, T., Syme, D. (2014). The F# Computation Expression Zoo. In: Flatt, M., Guo, HF. (eds) Practical Aspects of Declarative Languages. PADL 2014. Lecture Notes in Computer Science, vol 8324. Springer, Cham. 2014, https://doi.org/10.1007/978-3-319-04132-2_3

[13] Scala 3 Language Reference. cited 08.04.2024 https://docs.scala-lang.org/scala3/reference/

[14] Idris Language Reference. cited 08.04.2024 https://docs.idris-lang.org/en/latest/reference/index.html

[15] C# Asyncronious Programming Scenarious. cided 08.04.2024. https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-scenarios

[16] Shevchenko, R. Project Paper: Embedding Generic Monadic Transformer into Scala. Lecture Notes in Computer Science, 3401. 2022. https://doi.org/10.1007/978-3-031-21314-4_1

[17] dotty-cps-async. Retrieved 09.04.2024 https://github.com/rssh/dotty-cps-async

[18] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. Capturing Types. ACM Trans. Program. Lang. Syst. 45, 4, Article 21 (December 2023), 52 pages. https://doi.org/10.1145/3618003

[19] Koltin coroutines guide. Retrieved 09.04.2024 https://kotlinlang.org/docs/coroutines-guide.html

[20] Gears. An Async library for Scala. (cited 09.04.2024). https://github.com/lampepfl/gears