Kotlin Coroutines: Design and Implementation

Roman Elizarov
roman.elizarov@jetbrains.com
JetBrains
Saint Petersburg, Russia

Mikhail Belyaev
belyaev@kspt.icc.spbstu.ru
JetBrains Research
Peter the Great St. Petersburg Polytechnic University
Saint Petersburg, Russia

Marat Akhin
akhin@kspt.icc.spbstu.ru
JetBrains Research
Peter the Great St. Petersburg Polytechnic University
Saint Petersburg, Russia

Ilmir Usmanov
ilmir.usmanov@jetbrains.com
JetBrains GmbH
Munich, Germany

Abstract

Asynchronous programming is having its "renaissance" moment in recent years. Created in the 1980s, it was in use for quite some time, but with the advent of multi-core processors, it has been sidestepped by multi-threaded programming, which was (for a long time) the de facto standard of performing concurrent computations. However, since the 2000s, more and more programming languages have begun to include the support for asynchronous programming, some built around asynchronicity from the start, others including it later in their evolution.

In this paper, we explore the design and implementation of asynchronous programming in Kotlin, a multiplatform programming language from JetBrains, which uses *coroutines* for asynchronicity. Kotlin provides a compact built-in API for coroutine support, thus giving a lot of implementation freedom to the developer; this flexibility allows to transparently support different flavours of asynchronous programming within the same language.

We overview existing approaches to asynchronous programming, zoom in and talk about coroutines in detail, and describe how they are used in Kotlin as the basis for asynchronous computations. Along the way, we show the flexibility of Kotlin coroutines, highlight several existing problems with asynchronicity, how they are fixed or worked-around in Kotlin, and also mention future directions asynchronous programming might explore.

Onward! '21, October 20–22, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9110-8/21/10…$15.00
https://doi.org/10.1145/3486607.3486751

CCS Concepts: • **Software and its engineering → Compilers**; **Coroutines**; **Control structures**; **Concurrent programming languages**.

Keywords: Kotlin, asynchronous programming, coroutines, continuations, language design

ACM Reference Format:
Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. 2021. Kotlin Coroutines: Design and Implementation. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '21), October 20–22, 2021, Chicago, IL, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3486607.3486751

1 Introduction

Efficient hardware utilization has been one of the main foci of programming for a very long time. To use modern processors efficiently, one's program (among other things) should support performing several computations at the same time: while some computations are doing useful work, others may be waiting for data or results of other computations.

Multithreading [38] has been the traditional way of doing multiple computations *in parallel*, when each such computation is executed in a separate thread. If a computation needs to wait for something, its thread is *blocked*, to free the processor resources, and is later *unblocked* when the needed results are ready. While such paradigm solves the problem, it is not without several drawbacks, such as programming complexity and performance degradation for IO-bound tasks.

Different flavours of *asynchronous programming* are an alternative to multithreading. Unlike multithreading, which is based on coarse-grained threads, asynchronous programming is implemented via fine-grained *suspendable* computations, which can more effectively interleave with each other (allowing for better *concurrency*). Many of the early programming languages [4, 14, 60] had support for asynchronous programming, but with the spread of multi-core processors it has been all but replaced with multithreading.

However, in recent years there has been a "renaissance" moment, as more and more programming languages begin

to again include support for asynchronous programming. This may be attributed to it being better suited towards modern application needs, when you have to do a lot of waiting, for user inputs, network packets, etc. Different languages support asynchronous programming in different forms, depending on the design goals and its evolution.

In this paper we explore the design and implementation of asynchronous programming in Kotlin [8], a multiplatform programming language from JetBrains, which uses *coroutines* for asynchronicity. Its asynchronous programming support is somewhat unique, as Kotlin provides a compact set of built-in asynchronous APIs, with most of the implementation being in the form of user-provided libraries. This allows to support different flavours of asynchronous programming within the same language, giving the developer a lot of flexibility.

The rest of the paper is organized as follows. We overview existing approaches to asynchronous programming and their problems in section 2 and focus on coroutines in section 3. The analysis of design and implementation of coroutines in Kotlin is done in section 4, with multiple examples of their flexibility shown in section 5. We then discuss the current limitations and open design questions in sections 6 and 7 respectively.

2 Approaches to Asynchronous Programming

Different attempts have been made over the years to provide programming languages with facilities to support asynchronous programming. Here we overview the most important ones, to give context for better understanding of coroutine design and implementation in Kotlin.

2.1 Callbacks

Callbacks are probably the most widely-adopted way of structuring programs with asynchronous computations. A *callback* is essentially a user-defined function (passed as function value, lambda, function pointer, etc.) given to a callback-aware API with the intent to be called later, when some condition is met (e.g., a result of asynchronous computation is ready). Callbacks may be used to support both synchronous (such as error handling) and asynchronous execution models. They are particularly ubiquitous in system programming as means to interact with asynchronous APIs provided by the operating system, such as Unix signals [30].

Callbacks are comfortably suited for expressing *single* asynchronous interactions in otherwise synchronous code. For more complex scenarios, however, callback-based frameworks are known to suffer from a complication of code structure disproportionate to the complexity of the logic that code expresses, commonly referred to as "callback hell" [25, 31, 35]. This is especially visible in languages with support for anonymous or local functions where callback hell results in highly nested code structures and/or large number of small named local functions, which are very hard to refactor and maintain. For an example on how callback-based APIs induce highly nested and complex code refer to listing 1.

Listing 1. "Callback hell" example

```
fs.listDirectory(target) { files ->
    for (file in files) {
        file.readText { text ->
            sendMessage(peer, text) { answer ->
                database.store(answer) { result ->
                    if (result.error) error(result.error
                        )
                }
                sendMessage(leader, 'done') { answer ->
                    log(answer)
                }
            }
        }
    }
}
```

Callbacks, however, have one very important advantage over other approaches to asynchronous programming: frameworks based on callbacks are much easier to implement and use without explicit support from the language.

2.2 Futures / Promises

Futures [10] or *promises* [24] (also sometimes referred to as *deferreds* or *tasks*) are a step above callback-based computations, working as special proxies to not-yet-completed results of asynchronous computations[1]. A basic, but complete set of operations one can perform on a promise includes two: check for completion and get the final result. Trying to get the result of not-yet-completed promise, depending on the implementation, either blocks the program execution until the result is ready or causes a runtime error.

The biggest problem of "basic" promises described above is their inability to express fully asynchronous computations in a straightforward manner: with the basic set of operations, one has to either synchronously block the execution to wait for each promise (defeating the whole purpose of asynchronous programming) or introduce their own implementation of an "event loop" that periodically checks currently active promises for completion. This complicates the user code by polluting it with boilerplate code.

An extension to the concept of promises is *promise pipelining*: the term was introduced in the Joule programming language [54], but the key idea was originally coined in [33]. Promise pipelining introduces composition primitives between promises, allowing for building composite results from different asynchronous computations without the need to explicitly wait for each result. These can include running arbitrary code as soon as the computation is completed or building a new result from several unfinished results, to be

[1] Although both terms are used interchangeably, it seems more prevalent to use the term *future* for a read-only proxy object and the term *promise* for a proxy object that can be *fulfilled*. From this point forward, we will use the term *promise* to describe both kinds of proxies.

Listing 2. Promise pipelining example

```
fs.listDirectory(dir)
  .thenApply { files ->
      files.map { file ->
          file.readText()
              .thenCompose { text ->
                  sendMessage(peer, text)
              }
              .thenCompose { answer ->
                  database.store(answer)
              }
              .thenRun { result ->
                  if (result.error) error(result.error)
              }
              .thenCompose { result ->
                  sendMessage(leader, 'done')
              }
              .thenApply { answer ->
                  log(answer)
              }
      }
      .let { allOf(it) }
  }
```

computed when they are ready. A seasoned functional programmer may notice that pipelining allows a promise to implement a functional *monad* with the added ability to use already existing monad-based code patterns.

From a programmer's standpoint, promise pipelining is a mixture between promise-based computations and callbacks, as the higher-order primitives used in most pipelining computations receive a callback which is a function to run as soon as all required results are ready. It does, however, allow to express "truly" asynchronous computations without explicit blocking or event loops, also partially mitigating the "callback hell" problem.

The same example we used for callback hell illustration, but rewritten to take advantage of promise pipelining, can be seen in listing 2. This example uses pipeline naming from Java 8 [56] standard library (namely, `CompletableFuture` which implements a pipelined promise); traditionally named monadic API would replace `thenApply` with `map` and `thenCompose` with `flatMap`. While the promise-based version of the function is longer, it is also better at expressing relations between operations and is less nested. It is, however, still cluttered with asynchronicity-specific function calls and unnecessary lambda literals.

2.3 *async*/*await*

A natural improvement to techniques such as callbacks and promises would be one blending the differences between synchronous and asynchronous code. Asynchronous code is usually harder to reason about than synchronous code, and the added verbosity of callbacks and promises (them being second-class citizens) does not help either. *async*/*await* is an approach which is free of these problems by bringing asynchronous programming as a first-class language citizen.

async/*await* is based around expressing asynchronous computations as two interconnected parts.

Listing 3. Guess URL locale C# example (*async*/*await*)

```
public async Task<CultureInfo> GuessWebPageLocale(Uri
    uri)
{
    string text = await new WebClient()
        .DownloadStringTaskAsync(uri);
    CultureInfo localeGuess = GuessLocaleFromText(text);
    return localeGuess;
}
```

- *async* is used to mark the code (function call, expression, code block) which is executed asynchronously;
- *await* acts as a barrier for one or more *async* blocks, ensuring their execution is finished.

The first widely used programming language which adopted *async*/*await* was C#, version 5.0 [22] back in 2012[2]. From C#, this feature can be traced back to F#, version 2.0 [51] with its *asynchronous modality* [52] for expressions, in turn inspired by the work on concurrency monad for Haskell [17].

Depending on the exact flavour of *async*/*await* supported in your language, you need to mark different things as *async* or *await*. For example, an asynchronous C# function, which guesses a web page locale, may look as shown in listing 3. In C#, *async*/*await* implementation marks the code as follows.

- *async* is a *function* modifier, specifying that a function supports asynchronous execution;
- *await* is a built-in operator, which waits for the completion of its asynchronous argument.

C# has two additional restrictions w.r.t. its *async*/*await* implementation. First, *await* operator may be used only in *async* functions. Second, *async* functions should have an *awaitable* return type, i.e., a return type with promise-like capabilities.

If we forget about these restrictions and take a look at the code itself, it looks almost as the traditional synchronous code; the only things hinting at its asynchronous nature are *async*/*await* keywords. This is one of the strongest points of *async*/*await* compared to other asynchronous programming approaches: it requires little to no boilerplate, is readable and easier to understand.

A lot of programming languages used the C# approach as an inspiration and follow its *async*/*await* implementation. JavaScript, TypeScript, Dart, Hack, Python, Rust — all these languages use *async* functions containing *await* operators, allowed on their respective flavour of an awaitable type. C++ 20 [48], on the other hand, does not have an *async* modifier, considering all functions with *await* expressions as such, whereas Kotlin [8] opted for an inverted convention, with *async* modifier (called *suspend*) and no built-in *await* operator. While we will return to this design decision and attempt to explain it in section 4.2, it is important to note that, despite these differences, both C++ and Kotlin implementations still belong to the *async*/*await* approach.

[2]The aptly named Async Community Technology Preview included *async*/*await* even earlier, in 2011.

Listing 4. Guess URL locale Go example

```
func fetchUrlAsString(url string, ch chan string) {
  res, _ := http.Get(url)
  defer res.Body.Close()
  body, _ := ioutil.ReadAll(res.Body)
  ch <- string(body)
}

func guessWebPageLocale(url string) string {
  text := make(chan string)
  go fetchUrlAsString(url, text)
  return guessLocaleFromText(<-text)
}
```

As C#-style *async*/*await* approach requires the *async* function to have an awaitable return type, one may consider *async*/*await* to be a simple syntactic sugar over promise pipelining. However, as soon as we consider complex control-flow structures (e.g., loops or exception handling), the promise-based desugaring (if it had been used in practice) would have become much more intricate and complex.

However, there already exists a notion of functions supporting asynchronous execution — coroutines [19, 37]. Introduced in 1958 by Melvin Conway, a *coroutine* is essentially a function which can suspend and resume its execution; exactly the feature required for *async* functions. Used quite extensively in the early years of programming, present in languages such as Simula [14] or Modula-2 [60], they became secondary with the advent of multi-threaded programming, which we talked about in the introduction, but are currently returning, as the basic block of asynchronous programming.

Coroutines are what powers *async*/*await* implementation in most programming languages, and Kotlin is not an exception; the exact flavour of coroutines used, however, may vary between languages. We describe coroutine fundamentals and discuss their distinctive features in Kotlin in sections 3 and 4 respectively.

2.4 Green Threads

An alternative approach to asynchronous programming is the one based on "green threads": lightweight threads (or processes) managed in user space instead of kernel space. The first language to introduce them was occam [32], created in 1983 and heavily inspired by the communicating sequential process (CSP) algebra [28]. Concurrent ML [43] extended CSP with *first-class synchronous operations*, as a useful abstraction to simplify asynchronous programming.

Currently, green threads are what powers asynchronous programming in Erlang [4], Go [5] and Stackless Python [3]. An example `guessWebPageLocale` Go implementation[3] is shown in listing 4, where `go` keyword is used to execute a function in a green thread, called "goroutine" in Go.

[3]For the sake of brevity, the implementation omits error handling.

While programs written using this style of programming resemble traditional multi-threaded programs, there are several important differences w.r.t. their asynchronous nature. First, green threads support *cooperative* (aka non-preemptive) multitasking [11], i.e., the program itself signals when it could be suspended. While it does have the danger of one misbehaving program blocking the progress, the advantage of more efficient switches between green threads (compared to native threads) usually outweighs the possible downsides.

Second, the preferred way of sharing data is CSP-like *message passing*. By providing built-in, efficient versions of message passing utilities, languages with green threads try to compete with the performance of shared memory approaches, reducing the number of potential problems at the same time. However, if used incorrectly, message passing can be as error-prone as shared memory [55].

If we were to compare green threads to *async*/*await*, we would find them to be dual to each other as asynchronous programming approaches. *async*/*await* implementations attempt to make asynchronous code as similar as possible to regular, single-threaded, synchronous code. Green threads, on the other hand, do the same thing, but towards traditional multi-threaded code. Whether one approach is preferable to another from the programmer's point of view is up for debate, but to support green threads efficiently, a programming language needs to be either build around them from the start (as Go and Erlang) or significantly tweaked (as Stackless). Most languages, which include support for asynchronous programming later in their evolution, for this reason opt for the *async*/*await* approach.

2.5 Asynchronous Programming Problems

Asynchronous programming is subject to the same problems as programming in general; at the same time, it does have a number of unique problems, not relevant for synchronous code. In this section we briefly discuss two problems, which, in our opinion, have the most influence on the design space for a programming language with asynchronous programming support: "function colouring" and error handling.

2.5.1 Function Colouring. An important point of difference between different implementations of asynchronous computations is whether a particular implementation supports a clear division between asynchronous and synchronous code which we will refer to as *code* (or *function*) *colours*. This is a common idiom that comes under different disguises and formalizations, but, in essence, it boils down to a number of simple rules.

- Each program function (or, in some instances, a code fragment) is assigned a particular *colour*: for the sake of this paper, let's use *red* and *blue*;
- Blue code represents synchronous computations and can be accessed from both colours of code;

- Red code represents asynchronous computations and can only be accessed from other red code;
- There are special constructs which allow calling red code from blue code; alternatively, the code entry function is red.

In general, code colouring can be applied to a number of different problems by providing more colours and more rules on how these colours interact. Section 7.1 contains a brief overview of these multi-coloured approaches. In the rest of this paper, we use the term "function colouring" w.r.t. the problems that arise from asynchronous computations only, using the colouring setup given above.

Code colouring is pretty similar to (and can be viewed as a particular case of) computations with coeffects [39] and the guarantees it provides are very similar to functional monads [59] (e.g., `IO` monad in Haskell [40]).

Why Does Code Colouring Matter for Asynchronous Computations? In principle, one can imagine a language where all code is red and all computations may be asynchronous. But in reality, allowing asynchronous computations in every function introduces a number of problems.

- Asynchronous computations introduce additional overhead compared to normal function calls;
- Most programming languages are synchronous by nature and introduce asynchronous constructs for a particular purpose. Lack of clear distinction between synchronous and asynchronous functions leads to program comprehension problems: one would need to look at implementation of every program entity involved in a particular function call, going as deep as it is needed, in order to understand whether it is asynchronous or not. This is especially problematic when asynchronous computations are used *together* with multithreading;
- Error handling and propagation in red code (either in the form of exceptions or other mechanism) is a particularly complex task (see section 2.5.2 for details). If all code is red, we will have to use advanced error handling for all code.

On the other hand, if one were to solve these problems, a "colourless" language would have the flexibility of adding code colouring if and where it is needed, as a separate library or a language extension. Project Loom [1, 7] aims to achieve this ambitious goal for the JVM platform.

2.5.2 Error Handling. Another important problem in asynchronous programming is *error handling*. When you have multiple asynchronous executions running at the same time, and one of them fails, how you recover and continue the execution is not as straightforward as in synchronous code.

In regular code one has a well-defined caller-callee relation: errors in the callee propagate *upwards* to the caller and are handled there or continue propagating further. In asynchronous code this relation is no more, and without it there is no clear point-of-responsibility of who processes which errors. We also may need to propagate the error handling *downwards*, e.g., to cancel unnecessary computations.

One may use facilities available in the programming language to create a stand-in error handling replacement: for callbacks we can provide special error handler, with promises one could use pipelining, *async*/*await* implementations usually support `try/catch` blocks. However, all these cases are ill-suited for downward error handling, which would require a lot of boilerplate code.

More principled approach to asynchronous error handling would incorporate both upward and downward error handling. We can mention two approaches which are used in current asynchronous programming. The first one, pioneered in Erlang, is based on *supervision trees*. It is based on explicitly arranging your asynchronous tasks in parent-child trees, which correspond to the desired upward/downward error propagation structure. In case of an error, one may choose to isolate, propagate or restart the affected task subtree.

The second approach, first mentioned in 2016 and named *structured concurrency* [50], proposes to transfer the idea of structured programming [15] to the asynchronous programming as follows: all tasks are connected with their origins, and if task A starts task B, the lifetime of B cannot exceed the lifetime of A. This scheme establishes the launcher-launchee relation instead of the caller-callee one, and describes how to propagate errors and cancellations downwards.

Compared to supervision trees, structured concurrency is less flexible, as the error handling strategy is fixed. However, it is also less verbose and corresponds nicely to the way asynchronous code is usually written. That is why structured concurrency is gaining a lot of attention in recent years, either in the form of structured concurrency libraries or as a complete language feature [12].

3 Coroutines: Building Blocks for Asynchronous Programming

In this section we discuss coroutines as the basic building blocks for asynchronous programs. We describe what a coroutine is, how it is related to *continuations*, and how *async*/*await* is usually implemented via coroutines.

3.1 What Is a Coroutine?

Despite being used in one way or another for 50+ years [19], surprisingly, there still is no universal definition of what a coroutine is. When one talks about coroutines, they usually mean something akin to a suspendable function: function which can suspend and resume its execution, preserving the state between suspensions. These properties have been highlighted as characteristic of coroutines as early as 1980 [34].

In [37], coroutines have been classified using the following axes, on which their implementations may differ.

Symmetric/Asymmetric Control Transfer. A *symmetric* coroutine A can suspend itself and resume the execution to an *arbitrary* coroutine B, meaning one can freely jump between different coroutines, i.e., control transfer between coroutines is symmetrical. An *asymmetric* coroutine (also known as semi-coroutine [20]) can suspend itself, but it always resumes execution to its *caller*, like regular function calls, i.e., control transfer between coroutines is limited to the coroutine hierarchy.

While it has been thought symmetric coroutines are more expressive than asymmetric ones, as they were used and implemented in languages such as Simula [14] or Modula-2 [60], as a matter of fact, they can be expressed via one another [37]. At the same time, symmetric coroutines are much harder to understand [21], as they allow for unrestricted control transfer between coroutines. This is the reason most current coroutine implementations are asymmetric, the only notable exception being Julia [13].

Stackful/Stackless Implementation. A *stackful* coroutine implementation supports suspension within *arbitrary* nested functions; when resumed, a stackfull coroutine continues its execution exactly from the last suspension point, restoring the original function call stack. A *stackless* implementation, on the other hand, can only suspend within itself; to achieve asynchronous execution of nested functions, they must also be coroutines.

An avid reader may notice a degree of similarity between stackful/stackless coroutine dichotomy and the function colouring problem we discussed in section 2.5.1. This is not a coincidence: if a language supports stackful coroutines[4], it has the option to use a single colour for all code; if one uses stackless coroutines, the requirement to support nested suspension prompts the need for two separate colours. However, such a distinction is not mandatory, as stackful languages may introduce several colours, for example, for performance purposes, and stackless languages may use a single colour by saying the code entry function is asynchronous (red).

The majority of modern languages use stackless coroutines, with Lua [21], Ruby [23] and Julia [13] being among the few languages with stackful coroutine support. While stackful coroutines are more powerful than stackless ones, stackless coroutines can match most (if not the same) capabilities via careful handling of nested coroutine calls. Stackful coroutines are also noticeably harder to implement efficiently, which is another limiting factor for their wide-spread use.

First-class/Constrained Support. When working with coroutines, one may want to tweak different aspects of how the asynchronous code is handled; whether this is possible depends on what kind of coroutine support we have. A language with *constrained* coroutine support does not allow the developer to manipulate coroutines explicitly, i.e., they are hidden under some kind of abstraction; most *async*/*await* implementations belong to this category. For example, if one wants to customize the *async*/*await* handling in C#, they are limited to the API surface provided by the language [53]; in other cases (JavaScript, Dart) there is no extensibility at all.

First-class coroutine support means one can access the coroutine directly, as a first-class entity: saved to a variable, passed to other functions, suspended or resumed on request (as in Lua or Julia). Having coroutines as first-class citizens in your programming language allows for additional expressiveness and flexibility, e.g., the possibility of implementing custom cooperative multitasking facilities. On the other hand, this expressiveness comes at a cost of additional code complexity.

[4]Light-weight threads from section 2.4 are another example of stackful primitives used for asynchronous programming.

Listing 5. Scheme *call/cc* example

```
(let
  ([answer (+ 40
    (call/cc
      (lambda (cont) (* (cont 2) "fail"))
    )
  )])
  (print answer)
)

; prints 42
```

3.2 Continuations and Coroutines

When we talk about coroutines, we cannot but also talk about *continuations*, as these two primitives have more in common than one could imagine. A continuation represents the *whole* control state at some point of the program execution, and can be invoked later to continue execution from that point. Described first in 1964 by van Wijngaarden [58] together with the *continuation-passing style* (CPS) transformation for Algol 60, they have been (re)discovered several times afterwards [44], before being implemented in Scheme [49] as *call/cc* (literally "call with current continuation").

A call to *call/cc* allows the programmer to access the current continuation at the invocation point and, when needed, continue the execution with *call/cc* returning the value passed to the continuation. A simple example of *call/cc* is shown in listing 5. In the example, an erroneous operation of multiplying by string "fail" never executes, because `(cont 2)` transfers the control out (continuing the program execution from the point of *call/cc* invocation), and the code successfully prints 42.

One may notice the similarities between a *coroutine* and a *continuation*: both abstractions represent a computation which can be suspended and later resumed. These abstractions even seem to be interchangeable, and that is true. Continuations have been shown to be able to implement different

flavours of coroutines [27], and vice versa [29, 37, 42]. Even more so, if we take a look inside the coroutine implementations of most programming languages, it turns out they are built using CPS [9, 41, 45], i.e., they are implemented on top of continuations. C#, F#, JavaScript, Dart, Lua (to name a few) use CPS when compiling code which uses coroutines.

However, if we are talking about programming language support, unlike coroutines, continuations failed to get any kind of traction, besides Scheme [49], Standard ML [36] and Ruby [23]. The main reason for this is believed to be the inherent complexity of the continuation-based code, and the difficulty of making it performant.

3.3 Coroutines: Summary

Our coroutine overview shows that, if one were to focus on the expressive power of the coroutine implementation, they would use a first-class, stackful, (a)symmetric implementation (known in the literature as *full* coroutines [37]), as in Lua, Ruby or Julia. If one were to also consider other criteria, such as implementation performance or code readability, more restrictive coroutine implementations may be of better use; *async*/*await* implementations are based on constrained, stackless, asymmetric coroutines. As we will soon see, finding the balance between expressiveness, performance and readability was the main goal of Kotlin coroutine design.

4 Kotlin Coroutines: Design and Implementation

After overviewing the existing approaches to asynchronous programming in section 2 and exploring coroutines as basic asynchronous primitives in section 3, we are ready to describe the design and implementation of Kotlin asynchronous programming facilities.

4.1 Goals

Kotlin has been created as a *pragmatic* language: a programming language useful for day-to-day development, which helps the users get the job done via its features and its tools. Its support for asynchronous programming follows the same lines, which translates into the following main goals [6].

Independence from Low-Level Platform-Specific Implementations. As Kotlin is a multi-platform language, supporting compilation to JVM, JavaScript and native binaries, building its asynchronous support on top of other implementations (e.g., futures in JVM) would create a lot of problems, if one were to consider interoperability between platforms.

Adaptability to Existing Implementations. Being a relatively new language, Kotlin takes additional care about interoperability with already existing code; e.g., working with Java code on the JVM platform as transparently as possible. Considering there may already be established approaches of working with asynchronous code on specific platforms (e.g., promises in JavaScript or non-blocking input/output in JVM), Kotlin should support seamless integration of such APIs.

Listing 6. Guess URL locale Kotlin example

```
suspend fun guessLocaleFromText(
        text: String): Locale {
    // locale detection implementation
}

suspend fun guessWebPageLocale(
        url: URL): Locale {
    val text = HttpClient().get<String>(url)
    val localeGuess = guessLocaleFromText(text)
    return localeGuess
}
```

Support for Pragmatic Asynchronous Programming. The advent and proliferation of *async*/*await* approach, compared to other styles of asynchronous programming, showed the importance of code readability. While it is less expressive compared to more powerful approach based on full coroutines, it covers most pragmatic use cases and also allows for better performance.

4.2 Design

Kotlin asynchronous programming is built around the concept of a *suspending* function, similar to *async* functions in *async*/*await*. Our running `guessWebPageLocale` example written in Kotlin is shown in listing 6. Suspending functions are marked with the `suspend` keyword at declaration site, however, their call sites *are not marked* as `await` or `resume`, i.e., calls to suspending functions are implicitly awaited.

This design can be explained by pragmatic asynchronous programming. First, by defaulting to "*async* is *await*'ed implicitly", Kotlin avoids the "forgotten *await*" problem, characteristic of other *async*/*await* approaches. Second, asynchronous code becomes indistinguishable from synchronous one, making for easier code comprehension.

> *Calling a suspending function looks and feels the same as calling a regular function.*

As in other *async*/*await* approaches, there is a limitation on how one may invoke suspending functions: they cannot be called from regular, non-suspending functions. To bridge the synchronous and asynchronous worlds, one uses the *coroutine builders*: non-suspending functions with suspending lambda parameter, which are responsible for creating and starting the corresponding coroutine. An example with `runBlocking` coroutine builder is shown in listing 7.

However, these coroutine builders are not provided as built-ins, they are *implemented* in Kotlin using a very compact, but nonetheless complete coroutine API. As we will demonstrate in section 5, this allows one to implement custom coroutine builders or wrappers for alternative asynchronous facilities.

> *Most coroutine-related functions are not built-in, but implemented in pure Kotlin.*

Listing 7. `runBlocking` coroutine builder example

```
fun main(args: Array<String>) {
    // Non-suspending world
    runBlocking {
        // Suspending world
        val locale = guessWebPageLocale(
            URL("https://kotlinlang.org")
        )
        println(locale)
    }
}
```

Listing 8. `Continuation` interface

```
interface Continuation<in T> {
   val context: CoroutineContext
   fun resumeWith(result: Result<T>)
}
```

Listing 9. Low-level coroutine API

```
fun <T> (suspend () -> T).createCoroutineUnintercepted(
    completion: Continuation<T>
): Continuation<Unit>

suspend fun <T>
    suspendCoroutineUninterceptedOrReturn(
        block: (Continuation<T>) -> Any?): T

fun <T> (suspend () -> T).
    startCoroutineUninterceptedOrReturn(
        completion: Continuation<T>): Any?
```

As one may have already guessed, Kotlin approach to asynchronous programming is based on *coroutines*, which are compiled to CPS with *one-shot continuations* [16]. Every suspending function (or suspending lambda) is associated with its continuation, generated by the compiler. A continuation interface is referenced in listing 8.

A continuation can be resumed (invoked) to continue its execution with the given result. A coroutine context is somewhat analogous to thread-local storage: it contains data needed during the coroutine's lifetime.

> *Compilation of suspending functions to continuations is the first part of coroutine support built into the Kotlin compiler.*

Accessing these continuations is performed using a limited number of low-level, built-in intrinsic functions, which form the complete coroutine API. It is then extended with additional functions, written in Kotlin, to improve usability. The complete[5] built-in API for working with coroutines is shown in listing 9.

Function `createCoroutineUnintercepted` is used to *create* a coroutine corresponding to its extension receiver suspending function, which will invoke the passed `completion` continuation when it finishes its execution. However, the function does not start the coroutine, to do that, one have to call `Continuation<T>.resumeWith` method on the newly created continuation object. Suspending function `suspendCoroutineUninterceptedOrReturn` provides access to the current continuation (not unlike *call/cc*). If its lambda returns special `COROUTINE_SUSPENDED` marker, it also *suspends* the coroutine. Together with `Continuation<T>.resumeWith` function, which *resumes* and *starts* a coroutine, these functions form a complete coroutine implementation [37].

> *Low-level coroutine intrinsics are the second part of coroutine support built into the Kotlin compiler.*

[5]For the sake of brevity, we omit function versions with explicit receiver.

Listing 10. Continuation interceptor

```
interface ContinuationInterceptor : CoroutineContext.
     Element {
    companion object Key :
        CoroutineContext.Key<ContinuationInterceptor>
    fun <T> interceptContinuation(
        continuation: Continuation<T>):
            Continuation<T>
    fun releaseInterceptedContinuation(
        continuation: Continuation<*>)
}

fun <T> Continuation<T>.intercepted(): Continuation<T>
```

An attentive reader may notice the coroutine API functions mention something about *interception*. Continuation interception is the last piece of the coroutine puzzle. If we want to control how coroutines are executed w.r.t. native threads, for example, when working with UI frameworks, we need a way to specify how a given continuation should be executed. This is done using continuation interceptors, their interface presented in listing 10.

A continuation interceptor is a part of coroutine context; if present in the context, it is used by the coroutine implementation to *wrap* a continuation, via a call to `Continuation<T>.intercepted`. Unlike suspending functions/lambdas and coroutine intrinsics, which require compiler support, *interception* is supported in pure Kotlin, and built into the standard library. `startCoroutine` and `suspendCoroutine` functions call `Continuation<T>.intercepted` under the hood. By using the appropriate interceptor, one may change the way coroutines are scheduled onto the native threads.

> *Continuation interception is the third part of coroutine support built into the Kotlin standard library.*

4.3 Implementation

This section covers the main implementation details of Kotlin coroutines. Please refer to [57] for the full details.

4.3.1 CPS Transformation. Each suspendable function goes through a transformation from normally invoked function to continuation-passing style (CPS). For a suspendable function with parameters $p_1, p_2, \ldots, p_N$ and result type T a new function is generated, with an additional parameter p_{N+1} of type `Continuation<T>` and return type changed to `Any?`. The calling convention for such function is different

from regular functions as a suspendable function may either *suspend* or *return*:

- If the function returns some result, this result is returned directly from the function as normal;
- If the function suspends, it returns a special value `COROUTINE_SUSPENDED` to signal its suspended status.

However, the compiler prevents the user from manually returning `COROUTINE_SUSPENDED`. When the user wants to suspend a coroutine's execution, they

1. access the coroutine's continuation object by calling `suspendCoroutineUninterceptedOrReturn` intrinsic or any of its wrappers,
2. store the continuation object to resume it later,
3. pass the `COROUTINE_SUSPENDED` marker to the intrinsic, which is then returned from the function.

Since Kotlin does not currently support union types, resulting return type is changed to `Any?`, so it can incorporate the function's original return type and `COROUTINE_SUSPENDED`.

> *The CPS transformation makes the implementation of* `suspendCoroutineUninterceptedOrReturn` *intrinsic very straightforward: it passes the synthetic continuation argument into the* `block` *argument and returns* `COROUTINE_SUSPENDED` *if needed.*

4.3.2 State Machine Implementation. Kotlin implements suspendable functions as *state machines*, since such implementation does not require runtime support. On JVM we have to generate state machines because of JVM limitations: there is no built-in support to save current frame (i.e., local variables and stack) and to restore it later, yet[6]. Thus, these limitations dictate the two-colour model of Kotlin coroutines — the compiler has to know which function can potentially suspend, to turn it into a state machine — as well as it being stackless, with continuation objects forming a pseudo call stack and holding local variable values.

Each suspendable lambda is compiled to a continuation class, with fields representing its local variables, and an integer field for current state in the state machine. Suspension point is where such lambda can suspend: either a suspending function call or `suspendCoroutineUninterceptedOrReturn` intrinsic call. For a lambda with N suspension points and M return statements, which are not suspension points themselves, $N + M$ states are generated (one for each suspension point plus one for each non-suspending return statement). This allows the compiler to minimize the amount of code generated compared to functional continuation-passing style [18] which typically involves generating several continuation objects per invocation. To illustrate the state machine generation, refer to appendix B.

[6]Project Loom [1] is working towards introducing these capabilities to the JVM platform.

Listing 11. resumeImpl method

```
fun resumeWith(result: Result<Any?>) {
  val outcome = try {
    val outcome = invokeSuspend(result)
    if (outcome == COROUTINE_SUSPENDED) return
    Result.success(outcome)
  } catch (e: Throwable) {
    Result.failure(e)
  }
  completion.resumeWith(outcome)
}
```

> *The state machine transformation turns sequential code into suspendable by putting each suspension point in its own state.*

4.3.3 Suspension and Resumption. As mentioned in section 4.3.1, CPS transformation turns the result type of a suspending function to `Any?`, since it returns either T or `COROUTINE_SUSPENDED` and Kotlin does not support union types. If the function returns the `COROUTINE_SUSPENDED` marker, its caller returns the marker as well, until the execution reaches a coroutine builder, a boundary between two colours. This forms the basis for suspension of Kotlin coroutines.

Listing 11 shows `resumeImpl` method, inherited by all compiler-generated continuations. When a programmer resumes a suspended coroutine by calling this function, it calls the generated `invokeSuspend` method, passing `result` argument as its parameter. This both resumes the coroutine and replaces the result value of a suspending call. If `invokeSuspend` suspends, `resumeWith` immediately returns, leaving the coroutine in suspended state. Otherwise, it resumes the completion object, and since the completion chain mirrors the suspending function call chain, the execution continues as if there was no suspension.

> *A coroutine suspends by hoisting* `COROUTINE_SUSPENDED` *marker through the call stack and resumes by calling completion chain's* `resumeWith` *methods.*

4.3.4 Error Handling. When a coroutine is suspended, then resumed and then throws an exception, its actual call stack is different from the suspending call stack. However, we need to pass the exception to the original suspending caller, so the error handling behaviour is the same as for coroutine execution without exception. Listing 11 demonstrates that: we catch the exception, wrap it in `Result` inline class and resume the completion object with the wrapped value. Listing B.2 shows how the compiler generates a call to `throwOnFailure`, which rethrows the stored exception. Then the rethrown exception is again caught in the `resumeImpl` and the cycle continues, until the execution reaches `try-catch` block written by programmer, or it finally ends up in the outermost builder continuation `resumeImpl` method. This mechanism of wrapping and rethrowing preserves the behaviour of exception hoisting through the call chain, and

also allows the exception to go to another thread. The latter is essential for structured concurrency.

4.3.5 Structured Concurrency.

The current implementation of structured concurrency [50] in `kotlinx.coroutine` uses `CancellationException` to cancel the suspended coroutines. An alternative way would be to use another marker object in addition to `COROUTINE_SUSPENDED` and check for it in the generated state machine. However, the structured concurrency is not built into the language and is a part of `kotlinx.coroutines` library. Since the library cannot change the generated code, it uses exception handling for cancellation support.

Structured concurrency is preferred to supervision trees, as it is less verbose and better fits the conventional way of writing Kotlin code [2]. If one needs the additional flexibility of supervision trees, however, they could be implemented as a separate library.

4.3.6 Not Covered in This Paper.

Several things about coroutine implementation are intentionally left out of scope for this paper.

- **Suspendable function inlining**: inlining in Kotlin is implemented in a platform-specific manner and is rather complex, while not influencing the coroutine design in any significant way;
- **Coroutine optimizations**: there are several performance optimizations done on state machines and continuations in specific circumstances;
- **Particular context implementations**: coroutine context is a generic interface allowing for different kinds of invocation strategies for Kotlin coroutines. Particular implementations for these may be found in supporting libraries, while Kotlin itself does not provide any.

5 Design Evaluation and Usage Examples

5.1 Migrating Existing Asynchronous API to Kotlin Coroutines

One of the key features of Kotlin coroutine design is the ease of adaptability: not relying on any particular execution model (be it underlying thread pool or event loop) allows to easily adapt any existing asynchronous API and integrate it into canonical Kotlin code. In this section we showcase such *adaptations* of different styles of asynchronous APIs to Kotlin.

Callback-Based Approaches. Adapting any existing callback-based framework to coroutines is pretty straightforward using coroutine intrinsics. In listing 12 we can see an example callback-based function `someLongComputation` and its adaptation (with the same name) done by acquiring and suspending the current continuation, and resuming it later from the callback.

Listing 12. Callback adaptation example 1

```
// callback-based version
fun someLongComputation(params: Params,
                        callback: (Value) -> Unit)
// suspendable version
suspend fun someLongComputation(params: Params): Value =
    suspendCoroutine { cont ->
        someLongComputation(
            params,
            { result -> cont.resume(it) }
        )
}
```

Listing 13. Callback adaptation example 2

```
// callback-based version
// (see java.nio.AsynchronousFileChannel)
fun <A> AsynchronousFileChannel.read(
    dst: ByteBuffer,
    position: Long,
    attachment: A,
    handler: CompletionHandler<Integer,? super A>
)
// suspendable version
suspend fun AsynchronousFileChannel.read(
    dst: ByteBuffer,
    position: Long
): Int = suspendCoroutine { cont ->
    read<Unit>(dst, position, Unit,
        object : CompletionHandler {
            override fun completed(bytesRead: Int,
                                   attachment: Unit) {
                cont.resume(bytesRead)
            }

            override fun failed(exception: Throwable,
                                attachment: Unit) {
                cont.resumeWithException(exception)
            }
        })
}
```

A more sophisticated example of an API that allows callbacks to receive and propagate errors is Java standard library method `read` of class `java.nio.AsynchronousFileChannel`. This method receives an object of type `CompletionHandler` which serves as a callback for both valid and erroneous situations. Its adaptation example is shown in listing 13. Note that the `attachment` parameter, used in callback-based API to pass the data to the handler, is not needed, as the coroutine-based API naturally handles this situation.

Promise Pipelining-Based Approaches. Pipelined promises, as mentioned in section 2.2, are partially similar to callbacks, enough to be adapted to coroutines in a very similar way. Take listing 14 as an example. The original function is based on Java standard library `CompletableFuture` which is a typical example of a pipelined promise implementation.

It is easy to see that this can be generalized even further, by providing a universal suspendable function to every `CompletableFuture` instance. Refer to listing 15 for the implementation. We call this function `await`, because it serves essentially the same role as the *await* keyword in traditional *async*/*await* implementations. Even more so, though these

Listing 14. Promise adaptation example

```
// promise-based version
fun someLongComputation(params: Params):
     CompletableFuture<Value>
// suspendable version
suspend fun someLongComputation(params: Params): Value =
    suspendCoroutine { cont ->
        someLongComputation(params)
            .whenComplete { result, exception ->
            if (exception == null) // completed normally
                cont.resume(result)
            else // completed with an exception
                cont.resumeWithException(exception)
        }
}
```

Listing 15. Universal `await` for `CompletableFuture`

```
suspend fun <T> CompletableFuture<T>.await(): T =
    suspendCoroutine<T> { cont: Continuation<T> ->
        whenComplete { result, exception ->
            if (exception == null) // completed normally
                cont.resume(result)
            else // completed with an exception
                cont.resumeWithException(exception)
        }
    }
```

examples are given for `CompletableFuture`, they are easily adaptable to any promise implementation that supports some form of pipelining.

5.2 Implementing Different Styles of Asynchronous Computations Using Coroutines

While being able to *wrap* different existing frameworks for asynchronous computations, Kotlin coroutines also allow to *write* asynchronous code in different styles.

***Promise-Based* async/await.** While the absence of an *await* keyword (as well as a dedicated promise type) in Kotlin does not stop it from covering most practical use-cases, one may argue the classic promise-based *async*/*await* approach is more flexible, thanks to its ability to control how asynchronous computations are run in parallel and when exactly they are awaited. Thanks to the expressiveness of Kotlin coroutines, it is possible to adapt Kotlin coroutines to this particular style of programming.

An example of the possible implementation is given in listing 16. The `future` combinator runs a coroutine and returns its result as a promise. For an example implementation of this combinator for `CompletableFuture`, refer to appendix A.1.

Programming with Channels. As described in section 2.4, Go, Stackless Python and Erlang employ a different style of asynchronous programming based on green threads and message passing. While Kotlin does not have built-in support for green threads, the same approach can be emulated as a Kotlin library using a dedicated coroutine context (to allow executing coroutines in parallel, as pseudo green threads).

Listing 16. Future-based computation example

```
val future = future {
    // create a Future
    val original = loadImageAsync("...original...")
    // create a Future
    val overlay = loadImageAsync("...overlay...")
    ...
    // suspend while awaiting the loading of the images
    // then run `applyOverlay(...)` when both are loaded
    applyOverlay(original.await(), overlay.await())
}
```

Listing 17. Fibonacci function using channels

```
suspend fun fibonacci(n: Int, c: SendChannel<Int>) {
    var x = 0
    var y = 1
    for (i in 0..n - 1) {
        c.send(x)
        val next = x + y
        x = y
        y = next
    }
    c.close()
}
```

Listing 18. Python generator example

```
def infinite_palindromes():
    num = 0
    while True:
        if is_palindrome(num):
            i = (yield num)
            if i is not None:
                num = i
        num += 1
```

A possible implementation of such a library on top of Java's `ForkJoinPool` is given in appendix A.2. An example of a function[7] continuously feeding Fibonacci numbers to a channel can be seen in listing 17.

Generators. A *generator* is a common name for the language constructs which allow producing lazy sequences of values from seemingly sequential code. While they are just an example of asynchronous programming with suspended states, generators are typically implemented as a completely separate language feature (as in C#, Python and JavaScript).

Take for example the Python code given in listing 18[8]. Despite looking as sequential (and non-terminating) code, this function actually produces an infinite sequence of "palindrome" numbers using the built-in `yield` keyword.

Kotlin does not provide any special language feature to support generators, as they can be implemented as a library on top of coroutines, with `yield` being a suspendable function. Same palindrome-generating function, but implemented in Kotlin, is given in listing 19; the example

[7] This example is modeled after example 4 from "Tour of Go" (https://tour.golang.org)

[8] This example is taken from "Introduction to Python: Generators" (https://realpython.com/introduction-to-python-generators)

Listing 19. Kotlin generator example

```
fun infinitePalindromes(): Sequence<Int> = sequence {
    var num = 0
    while (true) {
        if (isPalindrome(num)) yield(num)
        ++num
    }
}
```

implementation of `sequence` and `yield` is given in appendix A.3. Kotlin standard library provides a more complex and flexible implementation of these functions.

5.3 User Stories

We would also like to mention two stories of users adopting coroutines as the go-to way of asynchronous programming. First, after Android became "Kotlin first" since 2019, it also accepted coroutines as the "recommended solution for asynchronous programming"[9]. The developers mention increased productivity after switching to coroutines.

Second, Amazon recently shared their experience of developing Prime Video profiles using Kotlin[10]. Among other Kotlin advantages, they mention coroutines to be an improvement over futures for developing distributed systems. They also say structured concurrency "makes the application more robust and decreases resource waste".

They mention, however, erroneous blocking in a coroutine running in a default Dispatcher may consume all its available threads and create hard-to-spot bugs. Thorough testing and chaos engineering [46] help to deal with such problems.

They also conducted a satisfaction survey which shows mostly positive results w.r.t. coroutines (e.g., their readability and performance), but also mentions steeper learning curve and the danger of the above-mentioned blocking bug.

6 Current Limitations

While the current design of Kotlin coroutines is pretty flexible and allows to achieve all the goals given in the previous sections, there still is a number of problems and limitations that will have to be addressed in the future. Most of these are related to Kotlin-specific design and may not be fully applicable to other languages.

6.1 Colour-Transparent Functions

The interaction between function colouring (section 2.5.1) and higher-order functions is a rather complex one. It is sound and easy to distinguish blue and red functions, both anonymous and named, using the design described in section 4 in the complex cases, but surprisingly many *simple* functions present the following problem.

[9] https://developer.android.com/kotlin/coroutines

[10] https://aws.amazon.com/blogs/opensource/adopting-kotlin-at-prime-video-for-higher-developer-satisfaction-and-less-code/

Listing 20. `invokeAll` example

```
suspend fun invokeAll(
    collection: Collection<
                    suspend () -> Unit
                >
) {
    collection.forEach lambda@ { element ->
        element()
    }
}
```

Consider the example in listing 20. A very simple function `invokeAll` receives a collection of asynchronous computations and attempts to invoke them in sequence. It is, however, impossible due to the function colouring restrictions. Function `invokeAll` is red (denoted by the `suspend` modifier) as are all the computations in the collection. However, function labeled `lambda@` which is passed to `forEach` is blue. It is legal to call `forEach` from `invokeAll`, but it is illegal to call red computations from `lambda@` even though it is valid semantically.

One could make `forEach` and its parameter red, but that would disallow calling it from blue code. What we really need is a *colour-transparent* function that allows its argument function to be of the same colour as the calling context, as if it is called from that context directly.

If you try to write this example in modern Kotlin, however, it compiles and works as expected, because `forEach` is declared `inline`. Inlined higher-order functions allow many things, among them, they are *colour-transparent* for their inlined arguments. It is not a general solution, however: to declare a function `inline` one needs to adhere to strict requirements that may be undesirable (see [8] for details), while the only thing required for the function to be colour-transparent is to call its supplied function parameter immediately (i.e., in the same calling context) zero or more times.

Colour-transparency is a problem not restricted to asynchronous computations, but is inherent to function colouring in general. There are a number of possible design solutions to this problem, but it is yet to be thoroughly explored.

6.2 Platform Interoperability

Kotlin is a multiplatform language that (as of 2020) targets the following platforms: Java virtual machine, JavaScript transpilation and a number of native targets (through the LLVM infrastructure). In the future, this list can be extended to other possible targets. Some of these platforms already provide their own asynchronous facilities, some plan on doing so in the future. While the flexible design of Kotlin coroutines allows for adapting existing asynchronous primitives to be called from Kotlin, it is still an open problem for language designers to define how other languages targeting the same platform may access Kotlin coroutine-based APIs.

6.3 Coroutines and Object-Oriented Programming

The relationship between object-oriented programming and asynchronous computations is, somewhat surprisingly, quite a complex one. While it is certainly straightforward to define interoperability between functions and asynchronous constructs, it is much less so with other concepts frequently found in object-oriented languages, such as inheritance, constructors, destructors and polymorphism. Does it make sense to allow asynchronous constructors? How do coroutines fit with class hierarchies? Are synchronous and asynchronous functions to be allowed to mix? While there probably exists a solid and well-reasoned answer to these questions, Kotlin designers have not found it yet.

7 Open Design Questions and Future Work

The current design and implementation of coroutines in Kotlin gives inspiration to a lot of new ideas and possibilities. This section describes some of them and gives a glimpse on the directions asynchronous programming development may take in the future.

7.1 Function Colouring with More Colours

As described in section 2.5.1, function colouring is a concept that can be applied not only to asynchronous computations, but to other areas as well.

- One can use colours for separating UI-related code from business logics; it is a common pattern for UI frameworks to keep the UI-related code bound to a separate thread, thread pool or event loop, while making the programmer responsible for enforcing this binding. There are a number of works trying to solve this particular problem [26, 47], which may also be viewed as a case of function colouring;
- With a multi-platform language such as Kotlin it may be desirable to write both server-side and client-side code of a distributed application in the same code base, and distinguishing them can be done via colouring, which in this case should be a built-in language feature;

Summing this up, one can imagine a universal solution to code colouring problems: a design that is capable of introducing colours and interactions between them in a unified manner; the distinction between suspending and non-suspending functions then becomes an instance of this design. Whether such design exists (and is feasible to implement) remains an open research question.

7.2 Serializable Coroutines

One of the main points of modern coroutine design is the principal independence of coroutines and system threads / CPU cores: one coroutine may run on the same system thread or "jump" between threads while maintaining its basic properties of cooperative multitasking. A natural next step would be to make coroutines independent not only from program threads sharing the same memory, but from system processes or even physical machines.

One can imagine a single suspendable function being executed in the cloud, waiting for an asynchronous external process (say, a measurement coming from a remote sensor) to finish for days, then "woken up" absolutely transparently to the programmer, run on the next available resource in the cloud, and suspended until the next asynchronous result is required. One can also imagine a single asynchronous function starting on server and then transferring its flow to the client and continuing its job in the browser. Such "machine-agnostic" coroutines have lots of potential in the modern world, having possible applications to internet-of-things, cloud computing, etc.

8 Conclusion

In this paper we talked about how asynchronous programming is implemented via coroutines in Kotlin. We surveyed the current landscape of asynchronous programming support in different programming languages and focused on coroutines as one of the more flexible ways of allowing asynchronicity. We then discussed their design and implementation in Kotlin, and showcased the flexibility of implementing asynchronous facilities via a compact, but general API surface, by embedding different kinds of other asynchronous programming approaches (such as generators or channels) into Kotlin.

We also talked about two problems most important w.r.t. asynchronous programming, code colouring and error handling, and how these problems are tackled in Kotlin. As for open questions and future work, we discussed several limitations of the current coroutine implementation in Kotlin and mentioned some interesting directions the evolution of asynchronous programming might take in the coming years.

We believe our exploration of the asynchronous programming in Kotlin shows the benefits of providing the users with access to the underlying implementation details, as this gives them great flexibility in how to perform asynchronous programming. We also hope this paper to be a good starting point for those interested in Kotlin coroutines and their implementations.

Acknowledgments

To the Kotlin language team, for an awesome language. To the reviewers, for their time, comments, and suggestions which greatly helped to improve this paper.

A Example Implementations

A.1 The `future` Builder Implementation

```
class CompletableFutureCoroutine<T>(
    override val context: CoroutineContext
) : CompletableFuture<T>(), Continuation<T> {
    override fun resumeWith(result: Result<T>) {
        result
```

```
            .onSuccess { complete(it) }
            .onFailure { completeExceptionally(it) }
    }
}
fun <T> future(
    context: CoroutineContext = CommonPool,
    block: suspend () -> T
): CompletableFuture<T> =
        CompletableFutureCoroutine<T>(context).also {
             block.startCoroutine(completion = it) }
```

A.2 The Channel Implementation

```
import java.util.*
import java.util.concurrent.atomic.*
import java.util.concurrent.locks.*
import kotlin.coroutines.*

interface SendChannel<T> {
    suspend fun send(value: T)
    fun close()
    fun <R> selectSend(a: SendCase<T, R>): Boolean
}
interface ReceiveChannel<T> {
    suspend fun receive(): T // throws
         NoSuchElementException on closed channel
    suspend fun receiveOrNull(): T? // returns null on
         closed channel
    fun <R> selectReceive(a: ReceiveCase<T, R>): Boolean
    suspend operator fun iterator(): ReceiveIterator<T>
}
interface ReceiveIterator<out T> {
    suspend operator fun hasNext(): Boolean
    suspend operator fun next(): T
}

private const val CHANNEL_CLOSED = "Channel was closed"

private val channelCounter = AtomicLong() // number
     channels for debugging

class Channel<T>(val capacity: Int = 1) : SendChannel<T
     >, ReceiveChannel<T> {
    init { require(capacity >= 1) }
    private val number = channelCounter.incrementAndGet
         () // for debugging
    private var closed = false
    private val buffer = ArrayDeque<T>(capacity)
    private val waiters = SentinelWaiter<T>()

    private val empty: Boolean get() = buffer.isEmpty()
    private val full: Boolean get() = buffer.size ==
         capacity

    suspend override fun send(value: T): Unit =
         suspendCoroutine sc@ { c ->
        var receiveWaiter: Waiter<T>? = null
        locked {
            check(!closed) { CHANNEL_CLOSED }
            if (full) {
                addWaiter(SendWaiter(c, value))
                return@sc // suspended
            } else {
                receiveWaiter = unlinkFirstWaiter()
                if (receiveWaiter == null) {
                    buffer.add(value)
                }
            }
        }
        receiveWaiter?.resumeReceive(value)
        c.resume(Unit) // sent -> resume this coroutine
             right away
    }

    override fun <R> selectSend(a: SendCase<T, R>):
         Boolean {
        var receiveWaiter: Waiter<T>? = null
        locked {
            if (a.selector.resolved) return true //
                 already resolved selector, do nothing
            check(!closed) { CHANNEL_CLOSED }
            if (full) {
                addWaiter(a)
                return false // suspended
            } else {
                receiveWaiter = unlinkFirstWaiter()
                if (receiveWaiter == null) {
                    buffer.add(a.value)
                }
            }
            a.unlink() // was resolved
        }
        receiveWaiter?.resumeReceive(a.value)
        a.resumeSend() // sent -> resume this coroutine
             right away
        return true
    }

    @Suppress("UNCHECKED_CAST")
    suspend override fun receive(): T = suspendCoroutine
         sc@ { c ->
        var sendWaiter: Waiter<T>? = null
        var wasClosed = false
        var result: T? = null
        locked {
            if (empty) {
                if (closed) {
                    wasClosed = true
                } else {
                    addWaiter(ReceiveWaiter(c))
                    return@sc // suspended
                }
            } else {
                result = buffer.removeFirst()
                sendWaiter = unlinkFirstWaiter()
                if (sendWaiter != null) buffer.add(
                     sendWaiter!!.getSendValue())
            }
        }
        sendWaiter?.resumeSend()
        if (wasClosed)
            c.resumeWithException(NoSuchElementException
                 (CHANNEL_CLOSED))
        else
            c.resume(result as T)
    }

    suspend override fun receiveOrNull(): T? =
         suspendCoroutine sc@ { c ->
        var sendWaiter: Waiter<T>? = null
        var result: T? = null
        locked {
            if (empty) {
                if (!closed) {
                    addWaiter(ReceiveOrNullWaiter(c))
                    return@sc // suspended
                }
            } else {
                result = buffer.removeFirst()
                sendWaiter = unlinkFirstWaiter()
                if (sendWaiter != null) buffer.add(
                     sendWaiter!!.getSendValue())
            }
        }
        sendWaiter?.resumeSend()
        c.resume(result)
    }

    override fun <R> selectReceive(a: ReceiveCase<T, R>)
         : Boolean {
        var sendWaiter: Waiter<T>? = null
        var wasClosed = false
        var result: T? = null
        locked {
            if (a.selector.resolved) return true //
                 already resolved selector, do nothing
            if (empty) {
                if (closed) {
```

```
                wasClosed = true
            } else {
                addWaiter(a)
                return false // suspended
            }
        } else {
            result = buffer.removeFirst()
            sendWaiter = unlinkFirstWaiter()
            if (sendWaiter != null) buffer.add(
                sendWaiter!!.getSendValue())
        }
        a.unlink() // was resolved
    }
    sendWaiter?.resumeSend()
    if (wasClosed)
        a.resumeClosed()
    else
        @Suppress("UNCHECKED_CAST")
        a.resumeReceive(result as T)
    return true
}

suspend override fun iterator(): ReceiveIterator<T>
    = ReceiveIteratorImpl()

inner class ReceiveIteratorImpl: ReceiveIterator<T>
    {
    private var computedNext = false
    private var hasNextValue = false
    private var nextValue: T? = null

    suspend override fun hasNext(): Boolean {
        if (computedNext) return hasNextValue
        return suspendCoroutine sc@ { c ->
            var sendWaiter: Waiter<T>? = null
            locked {
                if (empty) {
                    if (!closed) {
                        addWaiter(
                            IteratorHasNextWaiter(c
                            , this))
                        return@sc // suspended
                    } else
                        setClosed()
                } else {
                    setNext(buffer.removeFirst())
                    sendWaiter = unlinkFirstWaiter()
                    if (sendWaiter != null) buffer.
                        add(sendWaiter!!.
                        getSendValue())
                }
            }
            sendWaiter?.resumeSend()
            c.resume(hasNextValue)
        }
    }
    suspend override fun next(): T {
        // return value previous acquired by hasNext
        if (computedNext) {
            @Suppress("UNCHECKED_CAST")
            val result = nextValue as T
            computedNext = false
            nextValue = null
            return result
        }
        // do a regular receive is hasNext was not
            previously invoked
        return receive()
    }
    fun setNext(value: T) {
        computedNext = true
        hasNextValue = true
        nextValue = value
    }
    fun setClosed() {
        computedNext = true
        hasNextValue = false
    }
}

    @Suppress("UNCHECKED_CAST")
    override fun close() {
        var killList: ArrayList<Waiter<T>>? = null
        locked {
            if (closed) return // ignore repeated close
            closed = true
            if (empty || full) {
                killList = arrayListOf()
                while (true) {
                    killList!!.add(unlinkFirstWaiter()
                        ?: break)
                }
            } else {
                check (!hasWaiters) { "Channel with
                    butter not-full and not-empty shall
                     not have waiters"}
                return // nothing to do
            }
        }
        for (kill in killList!!) {
            kill.resumeClosed()
        }
    }

    private val hasWaiters: Boolean get() = waiters.next
        != waiters

    private fun addWaiter(w: Waiter<T>) {
        val last = waiters.prev!!
        w.prev = last
        w.next = waiters
        last.next = w
        waiters.prev = w
    }
    private fun unlinkFirstWaiter(): Waiter<T>? {
        val first = waiters.next!!
        if (first == waiters) return null
        first.unlink()
        return first
    }

    // debugging
    private val waitersString: String get() {
        val sb = StringBuilder("[")
        var w = waiters.next!!
        while (w != waiters) {
            if (sb.length > 1) sb.append(", ")
            sb.append(w)
            w = w.next!!
        }
        sb.append("]")
        return sb.toString()
    }

    override fun toString(): String = locked {
        "Channel #$number closed=$closed, buffer=$buffer
            , waiters=$waitersString"
    }
}

// For the sake of page limit, implementation of
// locking, waiters and selectors is omitted.
// It can be found at:
//   https://github.com/Kotlin/coroutines-examples/blob/
    master/examples/channel/channel.kt
```

A.3 The sequence and yield Implementation

```
@RestrictsSuspension
interface SequenceScope<in T> {
    suspend fun yield(value: T)
}

@UseExperimental(ExperimentalTypeInference::class)
fun <T> sequence(@BuilderInference block: suspend
    SequenceScope<T>.() -> Unit): Sequence<T> =
    Sequence {
    SequenceCoroutine<T>().apply {
```

```
        nextStep = block.createCoroutine(receiver = this
            , completion = this)
    }
}

private class SequenceCoroutine<T>: AbstractIterator<T
    >(), SequenceScope<T>, Continuation<Unit> {
    lateinit var nextStep: Continuation<Unit>
    // AbstractIterator implementation
    override fun computeNext() { nextStep.resume(Unit) }
    // Completion continuation implementation
    override val context: CoroutineContext get() =
        EmptyCoroutineContext

    override fun resumeWith(result: Result<Unit>) {
        result.getOrThrow() // bail out on error
        done()
    }

    // Generator implementation
    override suspend fun yield(value: T) {
        setNext(value)
        return suspendCoroutine { cont -> nextStep =
            cont }
    }
}
```

B State Machine Examples

B.1 Multiple Suspension Points Example

```
val a = a()
val y = foo(a).await() // suspension point #1
b()
val z = bar(a, y).await() // suspension point #2
c(z)
```

B.2 State Machine Example

```
class <anonymous> private constructor(
    completion: Continuation<Any?>
): SuspendLambda<...>(completion) {
    // The current state of the state machine
    var label = 0

    // local variables of the coroutine
    var a: A? = null
    var y: Y? = null

    fun invokeSuspend(result: Any?): Any? {
        // state jump table
        if (label == 0) goto L0
        if (label == 1) goto L1
        if (label == 2) goto L2
        else throw IllegalStateException()
      L0:
        // result is expected to be `null` at this
            invocation
        a = a()
        label = 1
        // 'this' is passed as a continuation
        result = foo(a).await(this)
        // return if await had suspended execution
        if (result == COROUTINE_SUSPENDED)
            return COROUTINE_SUSPENDED
      L1:
        // error handling
        result.throwOnFailure()
        // external code has resumed this coroutine
        // passing the result of .await()
        y = (Y) result
        b()
        label = 2
        // 'this' is passed as a continuation
        result = bar(a, y).await(this)
        // return if await had suspended execution
        if (result == COROUTINE_SUSPENDED)
            return COROUTINE_SUSPENDED
      L2:
        // error handling
        result.throwOnFailure()
        // external code has resumed this coroutine
        // passing the result of .await()
        Z z = (Z) result
        c(z)
        label = -1 // No more steps are allowed
        return Unit
    }
    fun create(completion: Continuation<Any?>):
        Continuation<Any?> {
        <anonymous>(completion)
    }
    fun invoke(completion: Continuation<Any?>): Any? {
        create(completion).invokeSuspend(Unit)
    }
}
```

References

[1] 2018. Project Loom: Fibers and Continuations for the Java Virtual Machine. https://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html [Online; accessed 15/04/2021].

[2] 2018. Structured Concurrency. https://elizarov.medium.com/structured-concurrency-anniversary-f2cc748b2401 [Online; accessed 15/08/2021].

[3] 2019. Stackless-Python 3.7.5 documentation. (2019). https://stackless.readthedocs.io/en/v3.7.5-slp/stackless-python.html [Online; accessed 15/08/2021].

[4] 2020. Erlang Reference Manual. (2020). http://erlang.org/doc/reference_manual/users_guide.html [Online; accessed 15/08/2021].

[5] 2020. The Go Programming Language Specification. (2020). https://golang.org/ref/spec [Online; accessed 15/08/2021].

[6] 2020. Kotlin Coroutines. https://github.com/Kotlin/KEEP/blob/master/proposals/coroutines.md [Online; accessed 15/04/2021].

[7] 2020. State of Loom. http://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html [Online; accessed 15/08/2021].

[8] Marat Akhin, Mikhail Belyaev, et al. 2020. Kotlin language specification (version 1.4-rfc+0.3). https://kotlinlang.org/spec [Online; accessed 15/04/2021].

[9] Andrew W. Appel. 1991. *Compiling with Continuations.* Cambridge University Press.

[10] Henry C. Baker and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. *SIGPLAN Not.* 12, 8 (Aug. 1977), 55–59. https://doi.org/10.1145/872734.806932

[11] Joe Bartel. 2011. Non-preemptive Multitasking. *Comput. J.* 30 (2011), 37–39.

[12] Alan Bateman. 2020. Structured Concurrency. (2020). https://wiki.openjdk.java.net/display/loom/Structured+Concurrency [Online; accessed 15/08/2021].

[13] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. 2012. Julia: a Fast Dynamic Language for Technical Computing. *arXiv preprint arXiv:1209.5145* (2012).

[14] Graham M. Birtwistle. 1973. *Simula Begin.* Petrocelli.

[15] Corrado Böhm and Giuseppe Jacopini. 1966. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Commun. ACM* 9, 5 (May 1966), 366–371. https://doi.org/10.1145/355592.365646

[16] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. 1996. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation.* Association for Computing Machinery, 99–107. https://doi.org/10.1145/231379.231395

[17] Koen Claessen. 1999. A Poor Man's Concurrency Monad. *J. Funct. Program.* 9, 3 (May 1999), 313–323. https://doi.org/10.1017/S0956796899003342

[18] Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or without? Whatever. *Proc. ACM Program. Lang.* 3, ICFP, Article 79 (July 2019), 28 pages. https://doi.org/10.1145/3341643

[19] Melvin E. Conway. 1963. Design of a Separable Transition-Diagram Compiler. *Commun. ACM* 6, 7 (July 1963), 396–408. https://doi.org/10.1145/366663.366704

[20] Ole-Johan Dahl and C. A. R. Hoare. 1972. *Chapter III: Hierarchical Program Structures.* Academic Press Ltd., GBR, 175–220.

[21] Ana Lúcia De Moura, Noemi Rodriguez, and Roberto Ierusalimschy. 2004. Coroutines in Lua. *Journal of Universal Computer Science* 10, 7 (2004), 910–925.

[22] ECMA International. 2017. *Standard ECMA-334 — C# Language Specification* (5 ed.). https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-334.pdf

[23] David Flanagan and Yukihiro Matsumoto. 2008. *The Ruby Programming Language: Everything You Need to Know.* "O'Reilly Media, Inc.".

[24] Daniel P. Friedman and David Stephen Wise. 1976. *The Impact of Applicative Programming on Multiprocessing.* Indiana University, Computer Science Department.

[25] Keheliya Gallaba, Ali Mesbah, and Ivan Beschastnikh. 2015. Don't Call Us, We'll Call You: Characterizing Callbacks in JavaScript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–10. https://doi.org/10.1109/ESEM.2015.7321196

[26] Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. 2013. Java UI: Effects for Controlling UI Object Access. In *European Conference on Object-Oriented Programming.* Springer, 179–204.

[27] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. 1984. Continuations and Coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.* Association for Computing Machinery, 293–298. https://doi.org/10.1145/800055.802046

[28] C. A. R. Hoare. 1978. Communicating Sequential Processes. *Commun. ACM* 21, 8 (Aug. 1978), 666–677. https://doi.org/10.1145/359576.359585

[29] Roshan P. James and Amr Sabry. 2011. Yield: Mainstream Delimited Continuations. In *First International Workshop on the Theory and Practice of Delimited Continuations*, Vol. 95. 96.

[30] Andrew Josey. 2004. The Single UNIX Specification Version 3. *Open Group* (2004).

[31] Kennedy Kambona, Elisa Gonzalez Boix, and Wolfgang De Meuter. 2013. An Evaluation of Reactive Programming and Promises for Structuring Collaborative Web Applications. In *Proceedings of the 7th Workshop on Dynamic Languages and Applications.* 1–9.

[32] INMOS Limited and INMOS International. 1988. *OCCAM 2 Reference Manual.* Prentice Hall.

[33] Barbara Liskov and Liuba Shrira. 1988. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. *ACM SIGPLAN Notices* 23, 7 (1988), 260–267.

[34] Christopher D. Marlin. 1980. *Coroutines: a Programming Methodology, a Language Design and an Implementation.* Springer.

[35] Tommi Mikkonen and Antero Taivalsaari. 2007. Web Applications: Spaghetti Code for the 21st Century. (2007).

[36] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML: Revised.* MIT Press.

[37] Ana Lúcia De Moura and Roberto Ierusalimschy. 2009. Revisiting Coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2, Article 6 (Feb. 2009), 31 pages. https://doi.org/10.1145/1462166.1462167

[38] Gregory M Papadopoulos and Kenneth R Traub. 1991. Multithreading: a Revisionist View of Dataflow Architectures. In *Proceedings of the 18th annual International Symposium on Computer Architecture.* 342–351.

[39] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: a Calculus of Context-Dependent Computation. *ACM SIGPLAN Notices* 49, 9 (2014), 123–135.

[40] Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages.* 71–84.

[41] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2016. Dependence-Driven Delimited CPS Transformation for JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* Association for Computing Machinery, 59–69. https://doi.org/10.1145/2993236.2993243

[42] Aleksandar Prokopec and Fengyun Liu. 2018. Theory and Practice of Coroutines with Snapshots. In *32nd European Conference on Object-Oriented Programming (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:32. https://doi.org/10.4230/LIPIcs.ECOOP.2018.3

[43] John H. Reppy. 1991. CML: a Higher-Order Concurrent Language. In *Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation.* ACM, 293–305.

[44] John C. Reynolds. 1993. The Discoveries of Continuations. *Lisp Symb. Comput.* 6, 3–4 (Nov. 1993), 233–248. https://doi.org/10.1007/BF01019459

[45] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming.* Association for Computing Machinery, 317–328. https://doi.org/10.1145/1596550.1596596

[46] Casey Rosenthal and Nora Jones. 2020. *Chaos Engineering.* Vol. 1005. O'Reilly Media, Incorporated.

[47] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. 2016. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *NDSS.*

[48] Richard Smith et al. 2020. Working draft, standard for programming language C++, document N4868. *ISO/IEC JTC1/SC22/WG21* (2020). http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4868.pdf

[49] Michael Sperber, R Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised^6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19, S1 (2009), 1–301.

[50] Martin Sústrik. 2018. Structured Concurrency. (2018). https://250bpm.com/blog:71/ [Online; accessed 15/08/2021].

[51] Don Syme et al. 2012. The F# 2.0 Language Specification. https://fsharp.org/specs/language-spec/2.0/FSharpSpec-2.0-April-2012.pdf

[52] Don Syme, Tomas Petricek, and Dmitry Lomov. 2011. The F# Asynchronous Programming Model. In *Practical Aspects of Declarative Languages*, Ricardo Rocha and John Launchbury (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–189.

[53] Sergey Tepliakov. 2018. Extending the Async Methods in C#. https://devblogs.microsoft.com/premier-developer/extending-the-async-methods-in-c/ [Online; accessed 15/08/2021].

[54] E. Dean Tribble, Mark S. Miller, Norm Hardy, and David Krieger. 1995. *Joule: Distributed Application Foundations.* Technical Report. ADd03.

[55] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. 2019. Understanding Real-World Concurrency Bugs in Go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* Association for Computing Machinery, 865–878. https://doi.org/10.1145/3297858.3304069

[56] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. 2014. *Java 8 in Action.* Manning publications.

[57] Ilmir Usmanov. 2021. The Ultimate Breakdown of Kotlin Coroutines. https://ilmirus.blogspot.com/2021/01/the-ultimate-breakdown-of-kotlin.html [Online; accessed 15/04/2021].

[58] Adriaan van Wijngaarden. 1966. *Recursive Definition of Syntax and Semantics.* North Holland Publishing Company.

[59] Philip Wadler. 1995. Monads for Functional Programming. In *International School on Advanced Functional Programming.* Springer, 24–52.

[60] Niklaus Wirth. 1956. *Programming in Modula-2.* Ju Lin.