SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY — COMPUTER SCIENCE

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Extending Kotlin with Algebraic Effect Handlers

Gabriele Pappalardo

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — COMPUTER SCIENCE

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Computer Science

Extending Kotlin with Algebraic Effect Handlers

Erweiterung von Kotlin mit Algebraic Effect Handlers

Author:	Gabriele Pappalardo
Supervisors:	Prof. Dr. Helmut Seidl
Advisors:	Dr. rer. nat. Michael Petter
Submission Date:	16th of October 2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 16th of October 2023 Gabriele Pappalardo

Abstract

In recent years, *Algebraic Effect Handlers* have emerged as new computational models to manage computation effects in programming languages. This model is linked to the area of functional programming languages. The thesis will illustrate how it is possible to adapt algebraic effect handlers into the multiparadigm *Kotlin* programming language through the implementation of a library over *continuations*. This master thesis is the result of a collaboration with *JetBrains Research*, which provided valuable insights and support in implementing algebraic effect handlers in Kotlin.

Contents

This page is intentionally left blank.

1. Introduction

1.1. A Real-World Analogy

Programs need to interact with the *external environment*, whether it is a network or the local machine. The environment in which programs operate takes the name of '**world**'. However, when the program interacts with it, the programs may cause a chain reaction taking the name of **effects**. Therefore, it is reasonable to model effects within a program so that we know how to deal with them. Effects are *pervasive* in computer science; some examples of them are *input/output*, *concurrency*, *distributed networks*, *exceptions*, and *choices*. **Algebraic effect handlers** are a way to give meaning or interpretation to effects.

Algebraic effects and handlers *allow a clear separation* between defining functions that involve effects and the way those effects are interpreted. For example, giving the pseudocode below, we define an algebraic effect `ReadSetting`. This effect takes a string representing a configuration's setting (e.g., number of parameters of a model, number of threads, …), and when performed, returns a string value.

```
1 effect ReadSetting = (String) -> String
```

An effectful function performs the effect without knowing how it is implemented, granting **high modularity**. For instance, the effect above can be interpreted by a *handler* that always reads the default user's setting from a data structure in memory, or from a local file, or even over a network in an asynchronous way.

1.1.1. Algebraic Effects going Mainstream

Recently, algebraic effects and handlers are spreading around programming language communities and academics. The reason behind this interest is that algebraic effects and handlers solve several issues in functional programming domains, such as implementing *lightweight concurrency*, *generators*, and *coroutines*, and safer ways to address *error handling in programs* in *direct-style*, without using callbacks or incurring in monads.

The latest research results and the discovery of new efficient techniques to implement them, brought people from industry to adapt algebraic effects in their projects or to create new libraries taking great inspiration from them. To cite a few, we have **OCaml 5.0** that implements algebraic effect handlers within the language, after having done modifications in the run-time system. The **React Hooks**' design [Met] has been influenced by effect handlers. In addition, GitHub's **Semantic** library [Git], used in GitHub repositories editor to jump through code symbols, makes use of effect handlers to guarantee an efficient analysis of the code. Algebraic effect handlers were also used to create the **Pyro** [Bin+18] probabilistic programming language, used in artificial intelligence research.

This surge in popularity and the problem-solving potential of algebraic effects and handlers lead to the question: *does Kotlin have room for implementing algebraic effect handlers*? **Kotlin** is a multi-platform programming language, mainly developed by **JetBrains**. How algebraic effects and handlers should be defined to be idiomatic to the language? What problems algebraic effects and handlers can solve if implemented in Kotlin?

1.2. Contributions

This document focuses on its main goals in understanding whether Kotlin can benefit from the introduction of algebraic effects and handlers. Therefore, the main contributions provided by the thesis are the following ones:

- A Kotlin library *that implements algebraic effects and handlers.*
- The *robustness* of Kotlin's coroutines design and machinery is shown.
- An optional static analysis tool to analyze effectful functions and their call in the absence of an *effect system* (shown in Appendix B).

1.3. Thesis Structure

The rest of this document is organized as follows:

- **Chapter 2** we are going to introduce background requirements to understand what algebraic effects and handlers are, with a small introduction to the Kotlin programming language and its coroutine system.
- **Chapter 3** exposes the related works and current state-of-the-art of implemented algebraic effects and handlers, while exploring the language design space and

analyzing *effect-oriented* programming languages. At the end of the chapter, we introduce the chosen metrics to design our library.

- **Chapter 4** introduces the '**Kotlin Effects**' library, showing some examples and diving into implementation details.
- The **last chapter** summarizes the thesis' content, showing the results achieved and how future development of this work can proceed.

2. Background

This chapter provides the foundational knowledge necessary for subsequent chapters of this document. We start with the introduction of algebraic effects and their extension with effects handlers. Then, we are going to introduce the **Kotlin** programming language, what coroutines and continuations are, and why they are necessary for implementing algebraic effect handlers in Kotlin.

The chapter will not delve into the theoretical aspects of algebraic effects, but rather takes a more practical approach, as one of the advantages of Kotlin is its focus on *pragmatism*. To deepen the theory of algebraic effects and handlers, we suggest reading the following resources:

1. *Gordon Plotkin and Matija Pretnar* - "Handlers of Algebraic Effects".
2. *Gordon Plotkin and John Power* - "Algebraic Operations and Generic Effects".

2.1. Algebraic Effects

Algebraic effects, which we will refer to as simply "effects" for the sake of brevity, are recently new concepts in the field of programming languages. Originally, they were presented as algebraic theory to prove the semantics of computational effects [PP09] [Pre15] (e.g., raised exceptions in many languages). An **algebraic effect** is a *set of signature operations* [Lin14] of the form $l : A \to B$ where $l \in \Sigma^*$ is a string label naming the operation and A, B are the respective domain and co-domain. Figure 2.1 shows an example that defines an effect "state" to save and load values in a background storage (that is, a database, a text file, …).

$$\text{State} = \{\text{get} : (\text{String}) \to \mathbb{N}, \text{put} : (\text{String} \times \mathbb{N}) \to \text{Unit}\} \tag{2.1}$$

Figure 2.1.: Definition of a *State* effect containing two algebraic operations: *put* represents the saving of a pair string-key, and *get* will take as input a string and will return a stored value.

Each algebraic operation is *abstract*, i.e., when an effect is performed, its meaning is defined by an algebraic effect handler, giving a concrete implementation to it.

2.1.1. Effect Handlers

Algebraic effects handlers can be thought of as a generalization of the exception mechanisms seen in other famous programming languages, such as *C#*, *Java*, *Python*, etc.; but capable of *resuming* after an exception has been thrown. They allow modeling advanced control flow. As [Hil15] explains, *algebraic effect handlers* can be seen as generalized functions that take, as input, an *abstract computation* (i.e., an algebraic effect), and interpret the operations that could potentially be executed during the evaluation of that computation.

Effect handlers have gained a lot of popularity in some mainstream programming languages, such as **OCaml** [Siv+21]. There are other academic programming languages that introduce effects as first-class citizens; to cite a few, we have: **Koka** [Lei14], **Unison** [Com], and **Effekt** [BSO20]. Some of them will be analyzed in depth in the next chapter.

Listing 1 shows a program, written in pseudo-languages similar to Kotlin, that uses algebraic effects and handlers. In the example, just one effect is defined, *Choice*, that when performed will yield a *boolean* value. As explained in the previous paragraphs, to give meaning to an effect, we *install a handler*. In the example, we have installed a handler that always resumes the function with a `true` value, resulting in the number 42 always being printed.

```
// Defines a new effect of type Choice, yielding a Boolean when
↪  performed.
effect Choice = () ->  Bool

// This effectful function has a simple effect signature of type Choice
fun throwCoin(): Int / {Choice} {
    return if (Choice()) {
        42
    } else {
        18
    }
}

// Effectful functions can be invoked only when an effect handler
// has been installed.
handle {
    val result = throwCoin()
    println(result) // Will always print '42'
} with {
    Choice -> {
        resume(true)
    }
}
```

Listing 1: Handling an effectful function with an installed handler. Performing an effect of type `Choice` yields a boolean value. The effect's meaning changes according to the installed handler; it can either return a value (`true` or `false`), or a random value.

The reasoning on the evaluation process of an effectful function can be thought of as shown in Figure 2.2. In the first column, we have the execution of an effectful function within the installed handler (delimited by the `with` block). Once the effect is performed, we substitute the expression `perform(Choice)` with the value passed when invoking `resume`. Then, we continue to execute the program as normal.

```
handle {
    if (perform(Choice)) {
        println(42)
    } else {
        println(18)
    }
} with {
    Choice -> resume(true)
}
```

```
if (true) {
    println(42)
} else {
    println(18)
}
```

Figure 2.2.: Evaluation process of performing a *Choice* effect.

A program that makes use of algebraic effects and handlers can be structured in three main parts [BSO18], respectively:

- **Effect Signatures**: Declare effects that a function can possibly trigger.
- **Effectful Functions**: Those are functions that can call effects explicitly, that is, when performing an effect; or implicitly, that is, when invoking another effectful function having a subset of the current function effect signature.
- **Effect Handlers**: Assign a "*meaning*" to an algebraic effect. When an effect is performed, an effect handler will manage the effect as the developer defined it (similar to the `catch` blocks used with exceptions).

Deep and Shallow Handlers

Effect handlers semantics can be classified into *two types*: **shallow** and **deep** handlers [HL18].

In the context of *shallow handler* semantics, if an effect is invoked during the execution of an effectful function, the corresponding effect handler will not resume its execution once the effectful function has completed its computation. Conversely, in *deep handler* semantics, the handler persists beyond the conclusion of the effectful function, retaining its state. Therefore, the lifespan of a deep handler can extend beyond that of the effectful function, allowing for continued operation and state preservation.

```
handle {
    perform(Log("Shallow
    ↪ Handlers"))
    println("After effect...")
} with {
    Log(msg: String) -> {
        println(msg)
        resume()
        println("After effectful
        ↪ function")
    }
}
```

```
handle {
    perform(Log("Deep Handlers"))
    println("After effect...")
} with {
    Log(msg: String) -> {
        println(msg)
        resume()
        println("After effectful
        ↪ function")
    }
}
```

Figure 2.3.: The following Figure illustrates the main difference between *shallow* and *deep* handlers. Briefly summarizing, in shallow semantics (on the left), the handler's execution is not resumed when the effectful function is done. Thus, the line `After effectful function` will not be printed on the screen. In deep semantics (on the right), the handler will continue living when the effectful function is resumed. And once the effectful function is computed, the effect handler will continue immediately after the `resume` call-site. Now the line `After effectful function` will be written on the output stream.

Using the code shown in Figure 2.3, we can see the difference between *shallow* and *deep* semantics. Namely, in shallow semantics the output we get is the following one:

```
1 "Shallow Handlers"
2 "After effect..."
```

Conversely, using the *deep* semantics, we get:

```
1 "Deep Handlers"
2 "After effect..."
3 "After effectful function"
```

showing how the lifespan of handlers change between the explained semantics.

Lexically and Dynamically Scope

In addition, algebraic effect handlers differ by their *scope*, that can be **statically** or **dynamically** determined [Bie+19]. The difference recalls the one seen in variable scoping. In lexical (or static) scope, the resolution of a variable's name occurs within the lexical context of the current function and its parent scopes. In dynamic scope, the name resolution depends on the program's runtime, considering the last binding to the name to be resolved. We can draw an analogy from variable scoping with the effect handler's one. Instead of resolving a variable name, when an effect is performed, we have to find an appropriate handler able to *catch and handle* that effect.

Lexically scoped effect handlers are bound to the lexical scope in which they are defined. They are active only within that scope and any nested scopes within it. When an effect is triggered, the search for an appropriate handler begins in the current lexical scope, and proceeds upward through enclosing scopes until reaching the topmost scope.

Opposite to static ones, dynamically scoped effect handlers are not bound to the lexical scope; instead, they are active throughout the entire dynamic execution of a program. Therefore, when using lexical scoped effect handlers, the control flow of a program is easier to read and follow, encapsulating the nearest environment where the handler is defined. Dynamic scopes are more challenging to follow, since their environment depends on the current runtime information.

2.1.2. Effect System

In effect-oriented programming languages, effects are usually tracked with an **effect system**. An *effect system* [TG08] is a formal method that extends a type system, while allowing the programmer to reason about potential effects that can arise during the execution of effectful functions. In addition, an effect system checks, at compile time, if a performed effect is properly handled with an handler. When the effect system validates if all effects have been handled, we can say that the program is **effect-safe**.

Effects can be combined accordingly with the algebra of *row effects*, which contains an *extension* operation (usually denoted by the symbol of the pipe $|$) and a *neutral element* (denoted by empty angle brackets $\langle\rangle$). Given different effects $e_i, e_j, e_k \in E$, the semantics of the extension operation is defined in Equation 2.2.

$$
\begin{aligned}
\langle e_i \rangle | e_j &:= \langle e_i, e_j \rangle \\
\langle e_i, e_j \rangle | e_k &:= \langle e_i, e_j, e_k \rangle \\
\langle e_i \rangle | \langle \rangle &:= \langle e_i \rangle \\
\langle e_i \rangle | \langle e_i \rangle &:= \langle e_i \rangle \\
\langle e_j, e_i \rangle &:= \langle e_i, e_j \rangle
\end{aligned}
\tag{2.2}
$$

Thanks to the effects, the programmer is able to categorize the *pure* and *impure* functions, allowing a better understanding of the codebase and how the functions interact with each other.

2.2. The Kotlin Programming Language

Kotlin[1] is an open-source, statically typed, multi-platform, and multi-paradigm programming language developed mainly by **JetBrains**. The language can be compiled into several targets, such as: *Java Virtual Machine*, *Native Code* (*e.g.*, x86_64, aarch64), *JavaScript* and *Wasm*. Kotlin's development began in 2011, and in 2016 it reached its 1.0 release. Later, in 2017, **Google** recognized Kotlin as the main programming language to develop **Android** applications. The language aims to be as concise and expressible as possible, overcoming the verbosity brought about by the Java programming language.

In 2018, during the release of version 1.3, Kotlin introduced **coroutines** in the language. With coroutines, the developer can use *asynchronous* and *non-blocking* programming paradigms, which contributes significantly to improving the performance of the application. Coroutines offer the advantage of being easily integrated into preexisting codebases through their compatibility with existing async APIs, allowing for efficient reusability. Listing 2 shows a small example of asynchronous programming in Kotlin. The implementation of coroutines is achieved through a **continuation-passing-style** (CPS) transformation. To understand how coroutines work, we should introduce the concept of **continuations**, and why they are important.

2.3. Continuations

Typically, the control flow of a program is modeled with what we call the **direct-style** in the literature [Dan94]. Therefore, when a function is invoked, we build a relationship between the *caller* (who invokes a function) and the *callee* (the invoked function). When

[1] Kotlin main website: https://kotlinlang.org/.

```kotlin
suspend fun fibonacci() = sequence {
    // It will contains the n-th number of Fibonacci
    var a = 0
    var b = 1
    while (true) {
        // Returns the n-th Fibonacci number and suspend
        yield(a)
        val t = a
        a = b
        b = t + b
    }
}
// Prints a list containing the first five numbers of Fibonacci
println(fibonacci().take(5).toList())
```

Listing 2: Implementing Fibonacci numbers generator using Kotlin coroutines. Each call to `yield` suspends the function, returning the value to the callee. When the function is called again, the execution will continue from where it was left.

a function is called, the caller passes the control flow to the callee, and when the function terminates, it goes back to the caller. In **continuation-passing-style** (CPS), the control flow does not return to the caller, but is passed through the usage of another function, which is called *continuation*.

A **continuation** is a first-class program object that represents the remaining computation from a certain point in the execution of a program. Functions in continuation-passing style always take an additional argument, the continuation, that is, the code to be executed after the function completes its computation. For the reader who may be confused, it may be helpful to think of the continuation parameters as a *function callback* seen in other programming languages (e.g., JavaScript).

Continuations are essential for our goal; they allow the implementation of **coroutines**, which are the building block for our library described in Chapter 4. Coroutines *allow functions to be suspended and resumed later*. Using coroutines is possible to temporarily suspend an effectful function and invoke an algebraic effect handler. The handler can then possibly resume the execution of the suspended effectful function from where it left off, thanks to the use of continuations. The importance of continuations extends beyond

```kotlin
// `add` function in direct-style, we return the value immediately.
fun add(a: Int, b: Int): Int {
    return a + b
}

// `add` function in CPS. Instead of returning the value, we pass it
↪ down
// as parameter to its continuation k.
fun <R> addCps(a: Int, b: Int, k: (Int) -> R): R {
    return k(a, b)
}

fun main() {
    // Will print '42'
    addCps(21, 21) { sum ->
        println(sum)
    }
}
```

Listing 3: *Direct-Style* vs *Continuation-Passing-Style* in Kotlin. As we can see from the code, the direct-style `add` function immediately returns the flow to the caller. In the CPS, we pass the return value to the continuation as a parameter, invoking it.

our specific library and is relevant to other effect-oriented programming languages, such as OCaml, Effekt, and more.

Continuations can be categorized in two dimensions: *linearity* and *delimitation* [Kis] [BWD96]. We have three types of linearity:

1. *Linear*: These continuations can be invoked only once, and once invoked, they cannot be reused.

2. *Multishot*: Compared to linear, they can be invoked several times, without any restriction on the number of invocations. They are useful when you have to explore all the possible paths of the control flow.

3. *Affine*: Are similar to multishot continuations in that they allow multiple invocations. However, they impose a usage restriction, typically allowing a finite but limited

number of invocations (e.g., at most once). These are used by OCaml in their implementation of effect handler, discussed in Chapter 3.

When we consider the *delimitation*, we refer to the scope captured by the continuation, more precisely:

1. *Delimited*: continuations are scoped or delimited to a specific portion of code. They capture the control flow from the current point in code up to a designated delimiter (in literature, there exist several operators to delimit the scope, such as *shift*/*reset*). These continuations return a value, allowing them to be easily composed with other functions.

2. *Undelimited*: they capture the entire program's control flow from the point of invocation to the program's entry point. Up to this moment, we keep the information of the entire function call stack and program's state. They do not return any value.

Kotlin, by default, implements *undelimited linear continuations*. Nevertheless, it is possible to create a *delimited* version, but this lies outside the scope of our current goal.

2.3.1. Kotlin's Coroutines and Continuations

Thanks to the introduction of coroutines in Kotlin [Eli+21], the language exposes the `Continuation` interface to the user, along with a set of compiler built-ins to manipulate them. These built-ins can be found in the `kotlin.coroutines` package and are intended for use by library developers.

Before illustrating how Kotlin exposes the continuation interface to developers, it is essential to explain where the language uses it. This will allow us to understand the design choices made by the language designers. Continuations allow Kotlin to implement **coroutines**. Without digging into the literature on coroutines, they can be thought of *as functions that can be paused and resumed at a later point in time.*

Kotlin introduces a new *keyword* (`suspend`) to mark a function that can be suspended. These functions are then compiled in continuation-passing-style. Kotlin's coroutines are compiled down into finite-state machines, such that they can remember at which point the computation was suspended. Each call to a suspendable function within a `suspend` function marks a *suspension point*; each of these points corresponds to a state in the finite-state machine. Continuations are useful for resuming a suspended computation where it was left off.

The Kotlin documentation defines a continuation as follows: “Interface representing a continuation after a suspension point that returns a value of type T.”. The interface has one property, a **coroutine context**, and a main abstract function `resumeWith`.

The built-ins contained in the library are mainly three, listed and explained as follows:

- **Create** Coroutine: create a new coroutine given a suspendable function. This will instance a finite-state machine ready to run the first state.
- **Start** Coroutine: as the name implies, it triggers the execution of a coroutine, that is, it runs the finite-state machine from the entering state.
- **Suspend** Coroutine: this is the most important primitive, it allows the suspending of a coroutine. This function takes a lambda as a parameter, which exposes the continuation for the suspended coroutine. Listing 4 contains an example of how the primitive can be used and how to manipulate the exposed continuation.

Continuations in Kotlin are *linear* (or single-shot), that is, they can only be resumed once. If a user tries to resume a continuation several times, the run-time will throw an exception.

```
1  var suspendedCoroutine: Continuation<Int>? = null
2
3  suspend fun <R> suspendAndStore: R = suspendCoroutine { cont ->
4      suspendedCoroutine = cont
5  }
6
7  fun startCoroutine(f: suspend () -> Unit) {
8      // ...
9  }
10
11 fun main() {
12
13     val f: suspend () -> Unit = {
14         println("Before suspending")
15         suspendAndStore()
16         println("Continuing execution...")
17     }
18
19     startCoroutine {
20         f()
21     }
22
23     println("Coroutine suspended...")
24     // Resume the coroutine after the `suspendAndStore` call
25     suspendedCoroutine!!.resume(Unit)
26 }
```

Listing 4: How the `suspendCoroutine` primitive works. When the primitive is invoked, the current coroutine will be paused, and it can be resumed later using the exposed continuation (see line 25).

3. Language Design Space

In this chapter, we analyze the language design space for fitting algebraic effect handlers in Kotlin. The analysis is based on the current *state-of-the-art* programming languages implementing algebraic effects and handlers. We begin by exploring a small taxonomy of programming languages, both popular in the industry and in the research field. We analyze how those languages implement algebraic effects focusing on advantages and disadvantages. The results of the next chapter derive from this analysis.

It is worth noting that the literature presents *libraries* that allow the implementation of effects and handlers in existing programming languages. For example, we have a library for *Haskell* (see *effect-handlers*[1]), some for *C/C++* (see *libhandler*[2], and *cpp-effects*[3]), *Java* [BSO18] and even for *JavaScript* (see *effects.js*[4]). Among these libraries, the common element is **continuations**.

3.1. Analysis of 'Effect-Oriented' Programming Languages

The analysis presents three snippets of code of functional programming languages: *OCaml*, *Unison* and *Effekt*. We show how the *syntax* and the *usage of effects* differ between those languages, outlining common aspects between them. We start the analysis with OCaml, since it is a well-known language used in both industry and academics.

3.1.1. OCaml

OCaml[5] is a general-purpose functional programming language created in 1996. OCaml is an "impure" language, in the sense that, compared to *Haskell* for example, it allows the creation of reference values, permitting the modification of a specific cell of memory. In 2022, the new compiler version, 5.0, introduced significant changes in language runtime, removing the **global interpreter lock** and adding **algebraic effect handlers** [Rec].

[1] effect-handlers: a Haskell library. https://hackage.haskell.org/package/effect-handlers
[2] libhandler: dynamic library in C99 used in Koka. https://github.com/koka-lang/libhandler
[3] cpp-effects: an effect handlers library in C++. https://github.com/maciejpirog/cpp-effects
[4] Algebraic Effect Handlers in JavaScript. github.com/nythrox/effects.js
[5] OCaml main website: https://ocaml.org/

```
open Effect
open Effect.Deep

type _ Effect.t += Yield : int -> bool t

let rec naturals_gen (n: int) =
    match perform (Yield n) with
    | true -> naturals_gen (n + 1)
    | false -> ()

let print_ten_naturals () =
    try_with naturals_gen 0 {
        effc = fun (type a) (eff: a t) -> match eff with
            | Yield n -> Some (fun (k: (a, _) continuation) ->
                            let _ = print_int n in
                            continue k (n <= 10)
                        )
            | _ -> None
    }

let _ = print_ten_naturals ()
```

Listing 5: Algebraic Effect Handlers in OCaml printing on the screen the first ten natural numbers.

The Listing 5 shows how effect and algebraic handlers work within the language. Let us analyze the main design points:

- *No built-in effects*: effects in OCaml are defined extending the `Effect.t` type. Each effect is a function that takes a number of parameters and returns a parametrized value `t`. This will be useful to infer the type of return of the generic function `perform`.
- *Performing an effect*: an effect is triggered by invoking the `perform` function. When an effect is performed, the language run-time will find a handler capable of dealing with it.
- *No effect signature*: as we can see, the signature of the effectful function `naturals_gen`

does not state the effect performed within the function body. The function's signature is the following one [6]: `naturals_gen : int -> unit`.

- *No new keywords*: effects are handled using the library function `try_with : ('b -> 'a) -> 'b -> 'a effect_handler -> 'a`. The parameters are described as follows: the first parameter is the effectful function taking a parameter of type `b` and returning a value of type `a`, the second parameter is the argument for the effectful function, and the last parameter is an effect handler parameterized on the `'a` type, returning a value of that type.
- *Distinction between deep and shallow handlers*: OCaml implements both types of effect handler described in Section 2.1.1 (see `Effect.Deep`).
- *Resumption of functions*: when handling an effect, OCaml exposes the parameter `k` representing the *delimited continuation* before performing an effect. To continue the execution of the effectful function, we can use the `continue` primitive, that takes the continuation and the return value of the performed effect.

Dropping Multishot Continuations

An interesting aspect of OCaml's algebraic effect handlers is the approach with *continuations*. Effect handlers make use of *delimited linear continuations* [OCa] (see Section 2.3). In the introduction section of the paper "*A separation logic for effect handlers*" [VP21], the authors debated why linear continuations have been preferred over multishot: "*allowing continuations to be invoked more than once breaks certain fundamental laws of reasoning about program*". Simply speaking, developers are used to think that a function has several entry points and one single exit point. However, with multishot continuations, a code block can be entered once and exited twice, which breaks the conventional reasoning used in program reasoning. In addition, multishot continuations can lead to wrong resource management, for instance, risking freeing a resource more than one time.

3.1.2. Unison

Unison is a novel functional programming language with many '*future-ready*' features. The feature we are interested in is the support for **abilities**, which are algebraic effects [Uni]. The abilities in Unison are based on the *Frank* language [LMM17]. As done previously with OCaml, let us proceed in analyzing the Listing 6 to understand what design choices were made by Unison's designers.

[6] In OCaml generic types have a single *quote* before their name. They can be read as *a generic a/b*.

```
unique ability Yield where
    yield: Nat -> Boolean

naturalsGenerator : Nat -> {Yield} ()
naturalsGenerator n = match (yield n) with
    true -> ()
    false -> naturalsGenerator (n + 1)

printHandler : abilities.Request {Yield} r -> {IO, Exception} r
printHandler = cases
    { pure } -> pure
    { Yield.yield n -> resume } ->
        printLine (Nat.toText n)
        handle (resume (n >= 10)) with printHandler

printTenNaturals : () -> {IO, Exception} ()
printTenNaturals _ =
    handle (naturalsGenerator 0) with printHandler
```

Listing 6: Abilities in Unison and their handling. When the *Unison Code Manager* loads the following program, it can be run typing `run printTenNaturals` in the terminal. This program prints on screen the first ten natural numbers.

The syntax of Unison is inspired by Haskell. Reading the code in the Listing 6 may not be straightforward. Thus, we help the reader identifying the main points.

- *ability definition*: abilities (i.e., effects) in Unison are built-in into the language. Their definition is based on what we illustrated in Section 2.1. An ability is a set of function signatures labelled with a name. Therefore, Unison has an *effect system*.
- `handle e with h`: invoking an effectful function requires the definition of an effect handler. Handlers' signature are defined with the `abilities.Request` built-in type, which represents requests to perform the ability's operations [7].
- *resumption*: resuming an effectful computation in Unison requires reinstalling a handler, therefore, the language implements *shallow* effect handlers semantics.

[7] A more detailed guide on unison ability handlers can be found at the following link: https://www.unison-lang.org/learn/fundamentals/abilities/writing-abilities/.

Compared to OCaml, Unison has special built-in handlers already defined for functions that use an IO ability. These ability handlers are provided by the language's runtime.

3.1.3. Effekt

Effekt is a functional research programming language similar to Scala. As done with previous languages, let us analyze how effects and handlers are designed within Effekt.

```
effect Yield(n: Int): Boolean

def naturalsGenerator(): Unit / {Yield} = {
    var stop = false;
    var a = 0
    while (!stop) {
        stop = do Yield(a)
        a = a + 1
    }
}

def main(): Unit / {Console} = {
    try { naturalsGenerator() } with Yield { n =>
        println(n); resume(n <= 10);
    }
}
```

Listing 7: Algebraic Effect Handlers in Effekt. The `naturalsGenerator` function signature contains, between brackets, the possible effects that need to be handled to invoke the function correctly (this is similar to checked exceptions in Java).

- `do` *notation*: to perform an effect, the developers must trigger it using the `do` command. The `do` statement is analogous to OCaml's `perform` function.
- *effects system*: the language incorporates an effect system to type-check the performing of effects and if they are handled by the callers. Therefore, the *algebraic effects* are built in the language. Effectful functions, in their signature, have on the right of the return type a *set of effects* that the function callers need to handle (e.g., in the example show in the listing, who calls `naturalsGenerator` needs to handle the `Yield` effect).

- *effects are requirements*: compared to other effect systems that inform the system which effects can be triggered when computing a result, Effekt treats algebraic effects as requirements. Therefore, the signature of the function `naturalsGenerator` is read as: "the function computes a value of type Unit requiring a capability for `Yield` in its calling context". Therefore, similar to what happens with checked exceptions in Java, given two effectful functions f_1, f_2 with their respective effect signature $E_1, E_2 \in \mathcal{P}(\text{Effects})$, we can say that f_1 can call f_2 if and only if $E_2 \subseteq E_1$.

- *built-in handlers*: as for Unison, Effekt has built-in handlers within the runtime (e.g., the built-in *println* function signature contains the `Console` effect). It is possible to create '*user-defined handlers*'.

- *continuations are not exposed*: when resuming the execution of an effectful function, the handler's context allows us to call the `resume` function, rather than exposing the effectful function's delimited continuation. In Effekt, continuations are multi-shot, they can be performed more than once.

- *contextual effect polymorphism*: Effekt does not support first-order functions, like the previous languages we have seen so far. Thus, the language has second-order functions along with the concepts of *blocks* [8] (i.e., functions can be passed as parameters, but they cannot be stored in variables). The language introduces the concept of '*contextual effect polymorphism*'. Listing 8 defines an effect polymorphic function, that can be read as follows: "given a block `f` without any further requirements, the function `eachLine` does not require any effects". Therefore, the function `f` does not impose any restrictions on its callers within `eachLine`. But if the function `f` has effects, then they need to be handled at the call site of `eachLine`.

```
1 def eachLine(file: File) { f: String => Unit / {} }: Unit / {}
2
3 // The type of the passed block is `String => Unit / {Console}`, and
4 // the final call will be of type `Unit / {Console}`.
5 eachLine(someFile) { line => do log(line)}
```

Listing 8: Contextual Polymorphism in action with Effekt blocks.

[8] Kotlin also supports the concepts of block with *inline* functions. For more details, check the documentation: https://kotlinlang.org/docs/inline-functions.html

3.2. Fitting effects into Kotlin

After analyzing the landscape of programming languages with algebraic effects, we faced ourselves with a big question: *should we implement algebraic effects and handlers as built-ins in the Kotlin compiler, or implement them as a library*? The next subsection dives into the question, trying to highlight advantages and disadvantages of both approaches.

Language	Inudstrial	Continuations	Effects Impl.	Handlers
OCaml	Yes	Linear	Library + Runtime	UD, SH, DE
Unison	No	Multishot	Native	BI, UD, SH
Effekt	No	Multishot	Native	UD, DE

Table 3.1.: Comparison of Languages Implementing Algebraic Effects and Handlers. Legend: UD (User-Defined), BI (Built-ins), SH (Shallow), DE (Deep)

3.2.1. Design Decisions

The comparison matrix shown in Figure 3.1 drove us to make design choices for algebraic effects and handlers in Kotlin. As we can see, we focused our analysis in four main axes, asking ourselves the following questions:

1. *industrial*
    - Are the programming languages under consideration primarily used in *academic research* or *industry*?
2. *continuations*
    - Which kind of continuations are exposed to the effect handlers?
    - What are the advantages and disadvantages of having linear or multishot continuations?
    - How do they impact the user's reasoning on the program?
3. *effects implementation*
    - How algebraic effects are implemented in the languages?
    - What are the benefits of having native effects handlers? Or integrating them as a library?
4. *handlers*: as seen in Section 2.1.1, handlers have a big taxonomy.
    - Shallow or Deep Handlers? Lexical or Dynamically scoped?
    - What are the limitations of shallow handlers over deep handlers?

Analyzing Industrial Usage

Let us start the analysis, considering each point in order. First, the *industrial* axis tells us if the considered programming language is used in industry or not. The only language satisfying this column is OCaml, since Effekt and Unison are not production-ready. It is important to give a special focus to industrial languages, since their user base and community are larger, and there are more real-world applications to consider. This is especially important when languages introduce new features that can break previous program versions. In the case of OCaml, effect handlers have been retrofitted tweaking the compiler to use handlers, and with the support of the library, keeping legacy programs safe. We can apply the same reasoning in Kotlin, we do not want to disrupt all existing programs with breaking changes. However, research languages can provide novel approaches to the design and usage of algebraic effects and handlers.

Analyzing Continuations

Continuations are essential for effect handlers. Integrating multishot continuations opens up the possibilities of implementing nondeterministic path exploration and probabilistic programming, but, as mentioned in Section 3.1.1, their introduction can be problematic for novel users. Therefore, we preferred to stick with *linear* continuations like OCaml, as Kotlin should be easy to reason with.

Analyzing Handlers Implementation

Unison and *Effekt* implement effect handlers within the compiler. OCaml takes a different approach, implementing them as a library with some tweaking on the language's runtime. The two research languages have been designed with integrating algebraic effects from the beginning. Therefore, their implementation is based on the latest research for implementing algebraic effect handlers efficiently when taking the running time in consideration. OCaml did not have any mechanism to alter the control flow of a program other than exceptions, so the authors modified the language's runtime to bring the needed modifications. Unison and Effekt use effects heavily, requiring the user to learn about them almost immediately, while OCaml does not force their usage. In addition, implementing effect handlers as a library would allow us to experiment with more flexibility without deeply modifying the compiler internals.

Analyzing Effect Handlers

We conclude our analysis with effects handlers, which are the main core of our work. The comparison matrix shows several types of handlers, and if they can be user-defined or not. Each choice has implications to consider, our main focus is on the easiness of reasoning. Deep effect handlers are immediate to understand for newcomers. Conversely, shallow effect handlers, may not immediate to the user since when an effect is handled, a new handler must be installed when the effectful computation is resumed, and this reduces the language's pragmatism. In addition, shallow semantic is convenient when implementing *mutual recursion*. It is possible to implement deep effect handlers using shallow effect handlers, re-installing the handler when the computation is resumed.

Idiomatic Syntax

In addition to the technical aspects of implementing algebraic effects and handlers, we also considered the syntax and user experience in our design decisions. Among the languages we have seen so far, and in the literature provided from the thesis, there is a common pattern when using algebraic effects, typically denoted as `handle e with h`, where `e` is an effectful computation and `h` is the associated effect handler. Furthermore, languages like *Effekt* and *OCaml*, emphasize the call-site notation for performing effects, often employing `do` or `perform` to indicate effect invocation.

Thus, we believe that re-using these patterns can help users familiarize themselves with algebraic effect handlers. Listing 9 shows how the `handle` block is represented in Kotlin, remaining faithful to the language idioms. This consistency can enhance code readability and ease of adoption for both newcomers and experienced Kotlin programmers.

```
1 /* Declaration of an effect. */
2 effect Yield: (Int) -> Boolean
3
4 handle {
5     /* Effectful Function. */
6 } with {
7     /* Handler's Block. */
8 }
```

Listing 9: Hypothetical syntax for using algebraic effects and handlers in Kotlin.

Listing 9 shows a possibility on how effect declaration can be integrated in the language if we were to modify Kotlin's grammar. An effect can be declared using the `effect` keyword, followed by a name and a function signature type in Kotlin. However, the feasibility of this choice depends on the implementation strategies discussed in Section 3.2.1. Introducing this new grammar requires modifications to Kotlin's grammar and the syntax compiler pipeline (e.g., *lexer* and *parser*). This is not the case for the `handle` block, thanks to Kotlin's ability to create *domain-specific-languages* (DSLs). [9]

3.2.2. Possible Implementation Strategies

Implementing algebraic effects and handlers into Kotlin can be possible by tweaking the compiler front-end and back-end. The front-end should handle the introduction of new constructs that are idiomatic to Kotlin. For example, during the design phase of the project, we thought of implementing the effects' signature in functions, similar to what *Effekt* and *Unison* do. The handling of effects will be then statically checked when functional effects are invoked, performing pattern matching on raised effects. For the back-end, algebraic effect handlers require *delimited continuations*, which are already provided in Kotlin thanks to the coroutine machinery. However, introducing effect signatures into the language would have meant introducing an *effect system*, and therefore *extending* the current Kotlin-type system. At the time of writing, Kotlin is under heavy development due to its new upcoming release (2.0), which significantly changes the compiler inner components, with the APIs evolving time by time.

3.2.3. The Choices

After analyzing the current literature of algebraic effects and handlers implementation in effect-oriented programming languages, we made several key design choices for integrating effects into Kotlin as *a library*. The following list summarizes all the choices based on the previous subsections:

- **Implementation**: We decided to implement algebraic effects and handlers as a library, rather than built-ins in the Kotlin compiler. This approach allows us to experiment with more *flexibility*, without delving into the compiler internals and unstable APIs. The effect handlers can be implemented on top of Kotlin coroutines (introduced in Chapter 2), as explained in the next Chapter. In addition, implementing algebraic effects and handlers as a library ensures that we do not disrupt existing programs with breaking changes.

[9] *Type-safe* builders: https://kotlinlang.org/docs/type-safe-builders.html

- **Continuations**: We use the linear continuations as OCaml. This decision aims to keep the reasoning for the user easy, as introducing multishot continuations can be challenging for novice users.
- **Effect Handlers**: We allow users to define their effect handlers, but we do not provide any pre-defined ones. We use the *deep handlers semantic*, because they provide greater flexibility and power. In addition, deep semantics handler do not need the re-installation in an effectful scope, keeping Kotlin pragmatism high.
- **Idiomatic Syntax**: We decided to use a syntax that is consistent with other languages that use algebraic effects, such as *Effekt* and *OCaml*, while being idiomatic to Kotlin. This includes using a `handle e with h` pattern for using algebraic effects, where `e` represents an effectful computation and `h` is the associated effect handler. As we decided to implement algebraic effects in a library, we do not modify the compiler's front-end. Therefore, as explained in the next Chapter, effects are defined inheriting a special library's interface. To mark explicitly the invocation of an effect, we use the `perform` notation as function calls.

The implementation and usage of the *library* will be explained in the next chapter.

4. The Kotlin Effects Library

This chapter illustrates the implementation of algebraic effect handlers in the Kotlin programming language as a standalone library. The final version of the library is the result of several iterations that will be analyzed throughout this chapter. At the end, we will show how we evaluate and validate the library, reimplementing examples seen in the literature, highlighting the limitations of our library.

4.1. Iterations

Originally, the library implementation was inspired by *Effekt*. However, due to constraints presented later in the subsections, we decided to perform some modifications allowing simpler effect handling while improving the overall *quality-of-life* of a developer. Each iteration has its advantages and disadvantages. All library implementations make heavy usage of *coroutine* mechanisms to implement algebraic effect handlers, while *effects* are represented with Kotlin interfaces. Rather than dive into the code when analyzing the iterations, we prefer to give insights on how Kotlin constructs were used to represent effects and handlers.

4.1.1. First Iteration: Strictly typed

The first iteration is quite strict with type checking, guaranteeing *type-safety* when performing effects. An *effect* is defined as an interface (see Listing 10) with two types of parameters: `A` representing the input type when the effect is performed, and `R` representing the output type after handling an effect.

```
1 interface Effect<A,R> {
2     context(EH)
3     suspend fun <E: Effect<A, R>, EH: EffectHandler<A, R, E>>
      ↪  perform(input: A): R
4 }
```

Listing 10: Definition of the Effect interface in the first library implementation.

The interface only defines the `perform` function as suspendable. Marking the function as suspendable is necessary because when an effect is performed, the execution of an effectful function must be paused to be possibly resumed later in a point. In line 2 we can notice that the function can only be called within the context of a subclass of the `EffectHandler`[1]. Let us proceed with the definition of the abstract class `EffectHandler<A, R, E: Effect<A, R>` (see Listing 11). The abstract class has three type parameters, the same `A` and `R` as before, plus a type parameter `E` bounded by the effect interface.

```
1 abstract class EffectHandler<A, R, E: Effect<A, R>> {
2     var storedContinuation: Continuation<R>? = null
3     fun resume(input: R) = storedContinuation?.resume(input)
4     /* Function implementing Handler's logic. */
5     abstract fun handle(effect: E, data: A): Unit
6 }
```

Listing 11: Definition of the EffectHandler in the first library implementation.

Let us see how an effectful computation can be defined and used with the Listing 12. As we can see, writing an effectful function is not straightforward. We have to address several points about the invocation:

1. *type ceremony*: Kotlin's type inference engine is not powerful enough to infer some types passed to the `handle` block. This led to writing many types to invoke an effectful function, reducing Kotlin's pragmatism. Furthermore, this implementation has another huge limitation, discussed in the next subsection.

2. *suspend `call-site`*: invoking an effectful function requires the caller to be a *suspendable* function. Therefore, effectful function invocations could only live with the *suspendable* world, making their access impractical for synchronous code.

3. *handling effects*: the definition of an effect handler occurs within the `with` block. The `with` block is just a Kotlin function that takes a lambda as input. The lambda has two parameters, respectively: the instance of the performed effect and a value of type `R` representing the effect's input.

4. *effect definitions*: effects definition do not have a reserved keyword, as happens with other languages. However, this problem can be addressed by transforming the

[1] **Context Receivers** are an experimental feature of the Kotlin compiler. More information can be found in the following resource [Fou].

library into a Kotlin compiler plugin, introducing the `effect` keyword to define new effects.

5. *effects composition*: with this implementation, effects and their handler cannot be easily composed.

6. *shallow handlers*: when an effect handler is invoked, its lifetime will end immediately after the effect has been handled.

7. *handling one effect*: the definition of an effect handler allows the handling of only one effect. This can be problematic since an effectful function could perform several effects. The solution to this problem, for this specific implementation, would be creating a cascade of effect handlers, one after the other, with the specific effect to handle. However, this solution does not guarantee that each performed effect will be handled, due to the absence of an effect system. Subsection 4.1.1 shows how it is possible to dynamically dispatch a performed effect to the correct handler.

```
1  // Let us create an effect extending the Effect interface
2  object YieldInt : Effect<Int, Boolean>
3
4  typealias YieldIntHandler = EffectHandler<Int, Boolean, YieldInt>
5
6  suspend fun main() {
7      handle<YieldInt, YieldIntHandler, Int, Boolean> {
8          var i = 0
9          var stop = true
10         while (!stop) {
11             println(i)
12             i += 1
13             stop = YieldInt.perform(i)
14         }
15     } with { effect, value ->
16         resume(value <= 100)
17     }
18 }
```

Listing 12: Invoking an effectful function with first library implementation.

Dispatching Effects

As mentioned, it is possible to overcome the limitation of one effect handler per effectful function. However, the solution to this problem introduces more code to write, and does not scale well with Kotlin pragmatism. First, let us suppose we have a list of effect handlers specialized for a specific algebraic effect. These handlers are always parametrized to the given effect with type parameter. But, we need to store the information about the type of the handled effect. To do so, we can introduce a new field to the `EffectHandler` class, that is, the class type of the effect `E`: `KClass<E>`. This is necessary since Kotlin, as Java, implements type parameters with *type erasure*, so we do not have fully access to the type with generic code, unless we use *reflection* [2].

```
1 abstract class EffectHandler<A, R, E: Effect<A, R>> {
2     /* Previous fields... */
3     abstract val effectType: KClass<E>
4 }
```

Listing 13: Keeping information on the type of effect handled.

With this information stored, the dispatching logic becomes more straightforward. Having a list of installed effect handlers, we can filter the list by the `effectType` of a given handler, matching it with the `KClass` of the currently performed event. If the result is *empty*, it signifies that no handler is available for the effect, prompting us to throw an exception. If the filtered list has *multiple elements*, handler collisions occur, making it ambiguous which handler to invoke. In the case of a *single matching handler*, we dispatch the effect to that handler.

The proposed solution is not a silver bullet. We still have a lot of "type ceremony" to encode for each handler. See the code in Listing 14.

[2]The documentation about Kotlin reflection can be found in the following link: https://kotlinlang.org/-docs/reflection.html

```kotlin
object Read: Effect<Unit, String>

class ReadAlwaysBob : EffectHandler<Unit, String, Read>() {
    override val effectType: KClass<Read>
        get() = Read::class

    override fun handle(effect: Read, data: Unit): Unit {
        resume("Bob")
    }
}
```

Listing 14: Defining a new effect handler containing all the necessary information to correctly dispatch a raised effect of type `Read`. When this handler is installed on an effectful function, the performing of the effect `Read` will always yield the string `"Bob"`.

Furthermore, this approach does not support parametrized effects, as the Kotlin compiler restricts assigning parametrized effects to the `effectType` field. These limitations make this approach less scalable and more cumbersome to use, highlighting the need for a more flexible solution when dealing with multiple effects in Kotlin.

4.1.2. Second Iteration: Relaxing Types

The second implementation totally relaxed the type constraints, redefining the `Effect` interface into a sealed class [3]. The use of sealed classes allows the developer to perform pattern matching with the `when` expression. As we can see from the Listing 15, now the `perform` function is not parameterized on any of the type constraints. These modifications, along with the one performed on effect handlers, remove the 'type ceremony' problem. However, we are now using Kotlin as a dynamic-typed language, losing all type information at static time, but trusting the cast checking at runtime.
As done previously, we list the main points of this implementation:

1. *effects definition*: effects can still be defined by extending the `Effect` class.
2. *leveraging types*: effect handler are now able to handle more than one effect compared to the previous version (see Listing 16).

[3] **Sealed classes** are a mechanism to create restricted class hierarchies providing more control over inheritance.

```kotlin
sealed class Effect {
    // Defininining two new effects
    internal class Yield(val input: Int) : Effect()
    internal object Abort : Effect()
}

context(EffectHandler)
suspend fun perform(e: Effect): Any = ...
```

Listing 15: Definition of the `Effect` class and `perform` function in second iteration.

3. *`abort` and `forward`* functions: the new interface for effect handlers introduces two new functions. `forward` tunnels an effect to the *uppermost lexical scoped* handler (we are going to define the meaning in the next paragraph), while `abort`, as the name implies, cancels the execution of an effectful function returning to the `handle` block's call-site.

4. *introducing the `perform` function*: another *major change* is how the input of an effect is passed to an effect handler; this happens using the effect class constructor, and the smart casting allows its access before compilation.

5. *additional notes*: this implementation, as the previous one, implements *shallow handlers*.

```kotlin
interface EffectHandler {
    var storedContinuation: StatefulContinuation?

    fun handle(e: Effect): Any
    fun resume(input: Any): Unit
    fun abort(input: Any): Unit
    fun forward(e: Effect): Unit
}
```

Listing 16: Definition of `EffectHandler` on second library implementation.

The Listing 17 shows the library implementation in use.

```
handle {
    var i = 0
    var stop = false
    while (!stop) {
        println(i)
        i += 1
        stop = perform(Effect.Yield(i)) as Boolean
    }
    perform(Abort)
} with { effect ->
    return@with when (effect) {
        is Effect.Yield -> {
            resume(effect.input >= 100)
        }
        is Effect.Abort -> {
            println("Number generation stopped!")
        }
    }
}
```

Listing 17: Invoking an effectful function using the second library implementation. The following effectful function implements a natural number generator up to 100.

Forwarding an Effect

In the previous paragraph, we mentioned the possibility of forwarding an effect. Let us suppose that we have two effectful computations, one inside the other. It may happen that a performed effect is not handled by the innermost handler. Thus, to handle this case, we can *forward* an effect to the uppermost handler, taking care of the effect performed. The Listing 18 shows an example of an effectful nested computation.

```
handle hOutermost@{
    handle hInnermost@{
        val ret = perform(Effec.Yield(42)) as Boolean
        println(ret) // Will output `true`
    } with { e2 ->
        forward(e2)
    }
} with { e1 ->
    when (e1) {
        is Effect.Yield -> {
            resume(true)
        }
    }
}
```

Listing 18: Forwarding an effect in second iteration of the library. When an effect is forwarded, the runtime will look for its closest outermost handler: if it is found, then the handler will be responsible to handle the effect correctly; if not, an exception will be thrown, aborting the effectful function execution.

4.1.3. Combining Type-Safety and Readability

Having identified what are the limitations and advantages of the first two approaches, we now propose the last one. The reason for the third and last iteration tries to strike a balance between the strengths of the first two approaches while addressing their respective shortcomings.

The first iteration is *type-safe, but verbose*. The second one, does not use *type parameters at all*, and it can be *easily broken with unsafe casts*, causing exceptions to be thrown at runtime. But, it uses pattern matching with the `when` expression, resembling the languages analyzed in the previous Chapter. When analyzing the effect-oriented programming languages, we noticed how performing an effect is marked explicitly using the `perform` notation. We decided to inherit this behavior by defining a `perform` function that triggers a given effect to handle.

The effect handlers are implemented through *pattern matching* on the performed effect. The next section will illustrate our library and its implementation details.

4.2. The Library

The '**Kotlin Effects**' library implements effects and algebraic effect handlers within the language. The library hides implementation details and exposes import functionalities of algebraic effect handlers, such as resuming the execution of a function when an effect has been invoked. This section will focus on the library's constructs and their semantics.

```
handle<Unit> {
    var stop = false;
    var a = 0
    var b = 1
    while (!stop) {
        stop = perform<Boolean>(Yield<Int>(a + b))
        a = b
        b = a + b
    }
    println("done!")
} with { effect ->
    when (effect) {
        is Yield<*> -> {
            resume(it.value >= 100)
        }
        else -> unhandled()
    }
}
```

Listing 19: An effectful function in Kotlin using the Kotlin effects library. Kotlin lambda expressions do not need labelled return statements.

Kotlin allows us to define *domain-specific languages* (DSLs). The main library function is `handle`, which accepts a suspendable lambda. This lambda '*lives*' in the context of an effectful scope; therefore, it is possible to *perform* effects within it.

Following the example shown in Listing 19, let us introduce the new terminology used by the library. First, the `handle` block takes as input an **effectful function**. These functions can perform effects, invoking the `perform` method, because an effective function lives within an **effectful scope**. Second, we have the `with` block, which installs a lambda

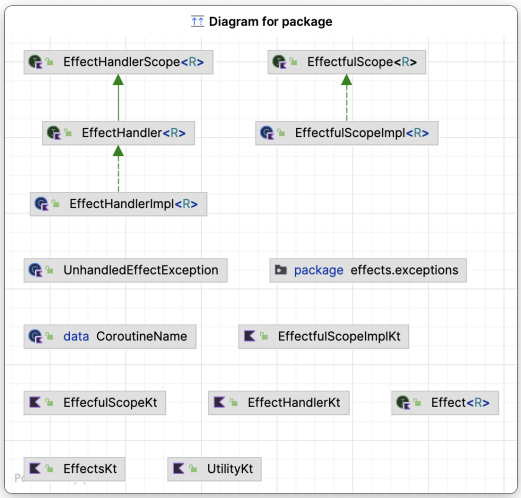

Figure 4.1.: Class and files defined within the Kotlin Effects library.

that will act as our effect handler.

The `unhandled` function plays a crucial role in our library's behavior. It is responsible for forwarding a performed effect to the next higher-level effect handler within the current scope. However, it is important to note that if no effect handler is capable of handling the effect, the library *raises* a runtime exception.

4.2.1. Implementation

The library works with the latest version of Kotlin (at the time of writing 1.9). It is made up of several classes and interfaces. This section will explain how they are related to each other and what their role is. The **code** for the library *can be found* in [Pap]. This explanation will not delve into details, but it will focus on the main implementation key points. Figure 4.1 shows the library main components.

Effect and Effect Handler Definitions

Let us introduce the main interface for defining new effects: `Effect<R>`. This interface defines only a type parameter `R` indicating what will be the return type of performing an effect. A user can simply define a new effect, creating a new class that extends the interface. The interface does not declare any method or property.

```kotlin
/* Definition of the Effect interface. */
interface Effect<R>

// The Read effect will yield a string when performed.
object Read: Effect<String>
```

Listing 20: The `Effect` interface and a sub-child defining the '*read*' effect.

The next interface under analysis is `EffectHandler<R>`, it defines two main methods:

- `fun <T> invokeHandler(effect: Effect<T>)`: it is used to start the execution of an effect handler with a given effect to handle.
- `suspend fun <T> resume(input: T): R?`: it is responsible for resuming the execution of an effectful function where it was stopped, that is, when performing an effect.

The interface defines a type parameter `R` representing the *return type* resulting from the execution of an effect handler.

```kotlin
interface EffectHandler<R>: Continuation<R> {
    fun <T> invokeHandler(effect: Effect<T>)
    fun <T> unhandled(): T = throw UnhandledEffectException()
    suspend fun <T> resume(input: T): R
}
```

Listing 21: The `EffectHandler<R>` interface, containing the resume and forward functions.

The last interface function, `resume`, is marked as `suspend`. Making the function as suspendable allows the programmer to define the behavior of an effect handler with *shallow* or *deep* semantics (see Section 2.1.1). n our implementation, we adopt the deep handler semantics. However, developers have the flexibility to define their own behavior by creating a new class implementing the `EffectHandler` interface. The interface extends the `Continuation<R>` interface, allowing to keep a coroutine context while the execution of an effect handler is paused. Furthermore, the interface defines the `unhandled` function, which allows to forward an effect to the nearest uppermost handler. An example of the usage of this function can be found in Section A.1.6.

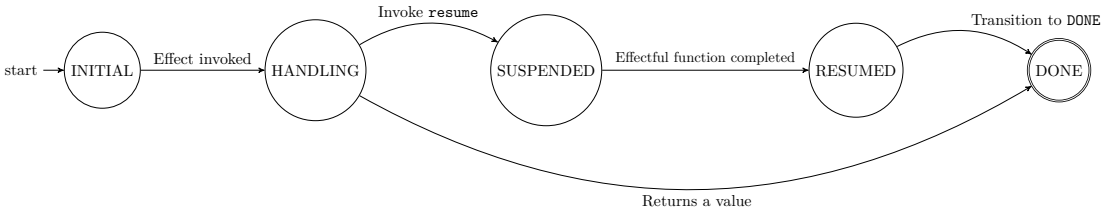

Figure 4.2.: Finite-State-Machine representing the status of an effect handler.

Implementing Deep Effect Handlers

Our implementation of the effect handler's interface, is given by the class `EffectHandlerImpl`. As previously discussed in Section 3.2.1, this class is responsible for implementing *deep handler semantics*. Deep semantics involve intricate state management within the effect handler, necessitating the tracking of various states throughout its lifecycle. To do so, we decided to model the lifecycle as a finite state machine. The Figure 4.2 shows how the automaton is modelled:

- `INITIAL`: This state signifies that the effect handler has not yet been invoked, and it is ready to start the execution.
- `HANDLING`: In this state, the effect handler is processing a performed effect. It is the phase where the core logic associated with the effect is executed.
- `SUSPENDED`: Transitioning to this state indicates that the `resume` function has been invoked, temporarily pausing the execution of the effect handler. The control flow is returned to the effectful function, where it suspended before performing an effect.
- `RESUMED`: The effect handler transits to this state when the effectful computation is done. This implies that the handler resumed the effectful function once, and therefore, it cannot be resumed again.
- `DONE`: The state denotes the completion of the effect handler's execution, indicating that it has finished its task.

In our implementation, when the `resume` function is called, the handler *temporarily suspends its execution* until the effectful computation is completed. Once the effectful function returns a value, the handler's execution resumes, and the `resume` function *yields* the returned value from the effectful function. However, if the handler returns a value without resuming the effectful function, it aborts the entire effectful computation and returns that value to the call-site of the effectful function. Listing 22 shows an example

```kotlin
object RandomInt: Effect<Int>

val x = handle h@{
    // `num` contains `42`
    val num = perform(RandomInt)
    return@h (num + 1)
} with {
    when (effect) {
        is RandomInt -> {
            val returned = resume(41)
            assert(returned == 42)
            returned
        }
        else -> unhandled()
    }
}

assert(x == 42)
```

Listing 22: The `resume` function returns the value of the effectful computation when it is done. The value yield by the `resume` function will be 42.

of the described behavior.

The `EffectHandlerImpl<R>` class depends on its *effectful scope*, explained in the next subsection.

The Effectful Function Scope

The main abstraction offered by the library is the `handle e with h` expression. However, behinds the scenes, this expression makes use of Kotlin's ability to model DSLs. At the core of the library's runtime logic lies the `EffectfulScope<R>` class (the class's declaration can be found in Listing 24). While this class cannot be manually instantiated, it can be constructed through an internally defined class builder, named `EffectfulScope.Builder<R>`. This builder serves the purpose of abstracting away implementation intricacies, ensuring a seamless and user-friendly experience for developers when utilizing effectful computations.

```
1 typealias EffectfulFunction<R> =
2     suspend (EffectfulScope<R>).() -> R
3 typealias EffectHandlerFunction<R> =
4     suspend (EffectHandler<R>).(Effect<*>) -> R
```

Listing 23: Definition of an `EffectfulFunction` and of `EffectHandlerFunction` types.

The creation of a new effectful function scope involves two essential components: an *effectful function* and the *logic of an effect handler*. In Listing 23, we define the types representing these components. Both types are aliasing suspendable lambda expressions. By making the lambda suspendable, it allows the computation to be paused when handling an algebraic effect using a handler. Notably, the `EffectfulFunction<R>` type specifies the `EffectfulScope<R>` as the receiver type, enabling the execution of algebraic effects through the perform function, as explained in the subsequent paragraphs.[4] The `EffectHandlerFunction<R>` type has the `EffectHandler<R>` as its receiver type, enabling the lambda to make calls to the `resume` function (explained earlier). Additionally, the lambda expression accepts a parameter of any subtype that inherits from the `Effect` interface, representing an algebraic effect performed by an effectful function.

Now that we have identified the essential components for an effectful scope, Listing 25 shows how `handle` and `with` calls are internally translated. As we can read, in Line 4 of the

[4]Kotlin receiver's object documentation can be found at the following link: https://kotlinlang.org/docs/lambdas.html.

```
class EffectfulScope<R> private constructor(
    private val effectfulFunction: EffectfulFunction<R>,
    private val effectHandlerFunction: EffectHandlerFunction<R>
) : Continuation<R>
```

Listing 24: Definition of the `EffectfulScope<R>` class. The class's constructor is private. Thus, an instance of the class can only be realized using the `EffectfulScope.Builder<R>` class.

Listing, the actual effectful computation is performed by invoking `runEffectfulScope` function, explained in the next section.

```
fun <R> handle(ef: EffectfulFunction<R>): EffectfulScope.Builder<R> =
    EffectfulScope.Builder(ef)

infix fun <R> EffectfulScope.Builder<R>.with(ehf:
↪   EffectHandlerFunction<R>): R =
↪   effectHandlerFunction(ehf).build().runEffectfulScope()
```

Listing 25: Definition of `handle` and `with` functions for creating effectful computations.

Effectful Scope Runtime

In this section, we dissect how the library's runtime works behind the scenes. As seen in Listing 25, when invoking the `with` function, we call the method `runEffectfulScope`. This method is the fundamental core of the library, and it manages the control flow between an effectful function and the defined effect handlers.

As for the effect handler's implementation, the `EffectfulScope<R>` class runtime is model through a slightly complex finite state machine, depicted by Figure 4.3.

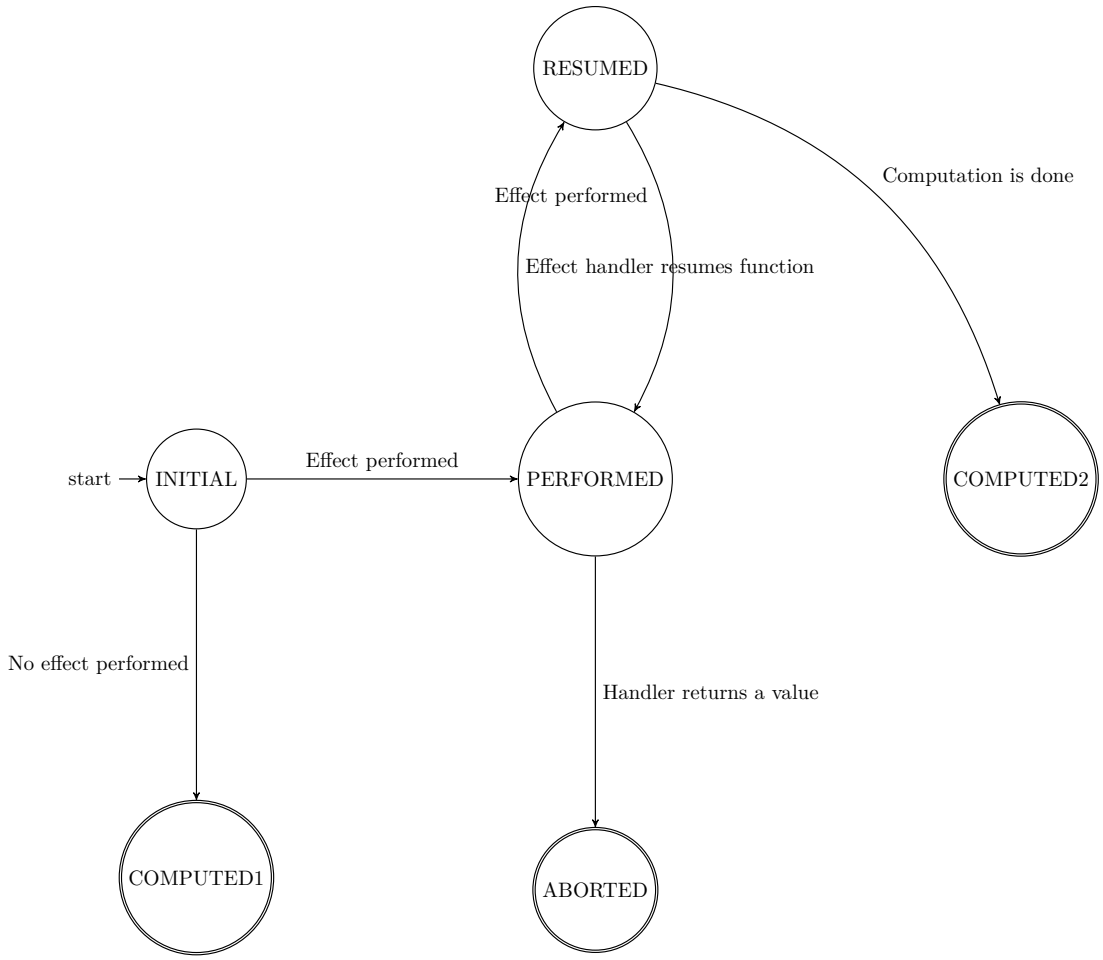

Figure 4.3.: Finite State Machine representing Effectful Scope runtime.

Let us analyze what are the main states and transitions. At the beginning, when the effectful function is running, the automaton lies in the `INITIAL` state. From this state we can have two paths:

1. The effectful function does not perform any effect, therefore, we can return the computed value, ending in `COMPUTED1`.

2. The function performs an effect that trigger the invocation of the associated effect handler's logic.

In the latter case, when an instance of the effect handler is created, it is added inside an internal stack, called the *effect handler's stack*. This data structure contains all the

suspended handlers that must be resumed when the effectful function computation is done. Thus, when an effect handler resumes the effectful function, the machine goes to the `RESUMED` state until a new effect is performed. If during the `RESUMED` state, no effect is performed, the machine transits to the ending state `COMPUTED2`. In this final state, the runtime propagates the computed result from the effectful function to the suspended handler's stack. Listing 26 shows how the paused effect handlers are pop-out from the stack.

```
var result: R = unwrapResult().getOrThrow()
while (!effectHandlerStack.empty()) {
    with(effectHandlerStack.pop()) {
        continueEffectHandlerExecution(result)
        result = unwrapResult().getOrThrow()
    }
}
return result
```

Listing 26: Algorithm showing how the stack of effect handlers is *unwinded*.

The illustrated algorithm is guaranteed to converge, since the number of handlers within the stack is finite. The first iteration passes the result computed by the effectful function to the latest invoked effect handler. Once the handler is resumed, its computed result will be propagated to the effectful scope, and therefore, will be passed to the next effect handler in the stack. Once the stack is empty, we can return the final computed value. To help the reader understand the control flow of the shown algorithm, we propose to visualize the Figure 4.4.

Performing an Effect

The class `EffectfulScope<R>` exposes the `perform` function, allowing effectful computation to perform algebraic effects. Listing 27 demonstrates the implementation of this function.

When an effectful computation initiates an effect, it undergoes suspension through the use of the `suspendCoroutine` primitive, as explained in Section 2.3.1. This suspension operation exposes the effectful function's *continuation*, which is stored in a private field, for subsequent resumption when an effect handler invokes the `resume` function.

```kotlin
object Empty: Effect<Unit>
val x = handle {
    perform(Empty) // EH0
    perform(Empty) // EH1
    perform(Empty) // EH2
    0
} with { effect ->
    when (effect) {
        is Empty -> {
            resume() + 1
        }
        else -> unhandled()
    }
}
assert(x == 3)
```

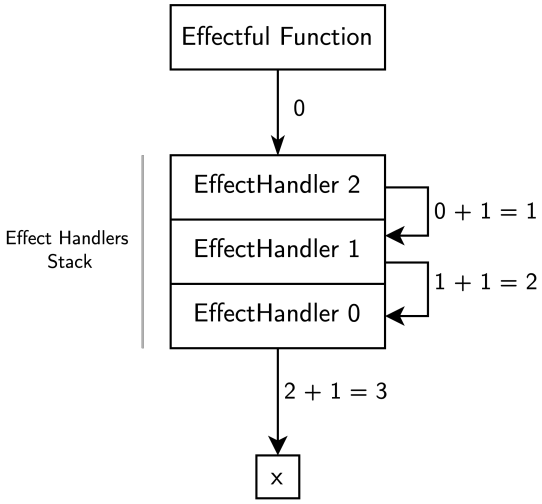

Figure 4.4.: Unwinding the stack of effect handlers when the effectful function computation is done.

```kotlin
suspend fun <T> perform(effect: Effect<T>): T {
    effectfulFunctionStatus = PERFORMED_EFFECT
    effectToBeHandled = effect
    return suspendCoroutine {
        // Suspend the effectful function and invoke the handler.
        effectfulFunctionContinuation = it as Continuation<Any?>
    }
}
```

Listing 27: Showing the implementation of the `perform` function. The shown code is abbreviated for readability reasons.

4.3. Validation & Evaluation

To evaluate and validate our algebraic effect handlers library, we designed a comprehensive validation process. The objective was to validate the expected behavior of the library in various scenarios, particularly focusing on replicating examples from several research papers. A part of the test suite can be found in Appendix A, showing some code examples with a proper explanation, highlighting the key points.

Our validation process yielded interesting insights into the capabilities and limitations of our library. Many examples and scenarios were successfully validated, aligning with our intended goals.

4.3.1. Faced Challenges

It is essential to note that not all examples find in literature were straightforward to validate. Specifically, we tried to imitate user studies from the *Effekt* webpage [5], but those examples rely on multishot continuations. Therefore, tests like *backtracking parsers*, *probabilistic programming*, *naturalistic DSLs*, and *non-deterministic path exploration* presented challenges in the validation process since they were not replicable.

4.3.2. Performances

Our main goal of the thesis is showing the feasibility of implementing algebraic effects within Kotlin. However, we performed small benchmarks on the overall space and time performances. The performed tests run generally fast. Table 4.1 shows the running time of the examples contained in Appendix A.

Test	First Time	Second Time	Third Time	Avg.
`No effects`	27ms	27ms	18ms	24ms
`Simple Effects`	27ms	27ms	18ms	24ms
`Abort effectful function`	36ms	27ms	27ms	30ms
`Pretnar - Simple Read Effect`	36ms	27ms	27ms	30ms
`Pretnar - Reverse Output`	18ms	18ms	27ms	21ms
`Pretnar - Collecting Output`	27ms	27ms	27ms	27ms
`State Effect - Emails Database`	36ms	54ms	36ms	42ms

Table 4.1.: Execution times of Appendix A test suite. Tests were run on a MacBook Pro with M1 Max CPU and 32GB of RAM. Slowest times cells are shaded.

[5] Effekt Case Studies: https://effekt-lang.org/docs/casestudies/

Slowdowns arise when we try to *resume* a considerable amount of continuations, as subsection 4.4.2 shows.

4.4. Library Downsides and Limitations

Our library is not absent from downsides; this section will analyze what are the major and minor problems, and how they could possibly be addressed.

4.4.1. Lack of an Effect System

One important *limitation* of our library is the *absence of an effect system*. In effect-oriented programming languages, effect systems play a crucial role in statically tracking and managing side effects. Without such a system, our library relies on runtime checks and type casting to ensure the correct handling of effects.

Type-Check Safety When Performing Effects

We decided to take a hybrid approach between the two iterations shown at the beginning of the chapter. However, this approach introduces a potential challenge related to type-check safety when performing effects. Specifically, there is a scenario in which the type of the returned value from an effect might not match the expected return effect value, see Listing 28. When the code shown in the Listing 28 is executed, an exception will be thrown at runtime, since the value `x` is expecting a `String` value, but `perform` yields an `Int`. Experienced Kotlin developers may suggest changing the function signature of `resume`, such that the given type parameter is *reified* (i.e., we keep track of the return type of the given effect). However, it is not possible to use *reified type parameters* when the function is marked as suspended.

```
object Read: Effect<String>

handle {
    val x: String = perform(Read)
    // The line below won't be reached.
    assert(false)
} with { effect ->
    when (effect) {
        is Read -> {
            // Cause a runtime exception.
            resume(42)
        }
        else -> unhandled()
    }
}

```

Listing 28: Showing how performing an effect can cause a runtime exception.

4.4.2. Runtime Efficiency

The heavy usage of coroutines in our library can lead to increased memory usage and invocation overhead, potentially impacting runtime efficiency. When an effect handler is invoked, a new coroutine is constructed. The instance of an effect handler is not reclaimed by the garbage collector until the stack of handlers is not empty. Using the '*IntelliJ Profiler*' we tested the running time of performing the most basic effect several times with the simplest handler (see Listing 29). In our test with the profiler, we ran the function several times with the following N values $(100, 1000, 10000, 100000)$, obtaining the chart shown in Figure 4.5. The plot shows that the running time scales *linearly* with respect to the number of performed effects. In real case scenarios, the number of performed effects would have lower orders of magnitude compared to the values seen in the test.

4.4.3. Other minorities

This section highlights other minor issues about our library. It is important to notice that those problems are common to all libraries' implementations.

```kotlin
@Test
fun `Performing a lot`() {
    val N = ...
    handle {
        for (i in (0..N)) {
            perform(Empty)
        }
    } with {
        resume(Unit)
    }
}
```

Listing 29: Unit test function used to estimate the performance of the library runtime.

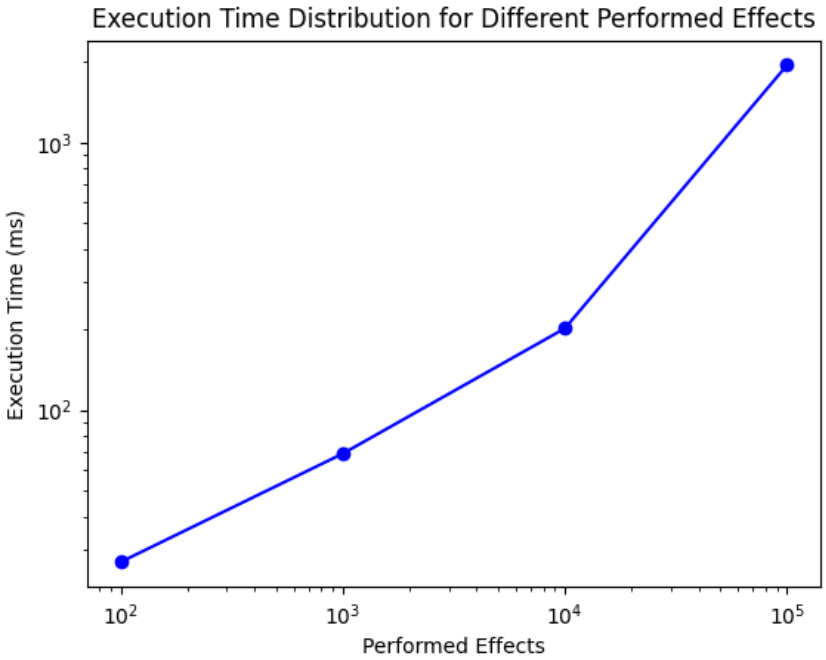

Figure 4.5.: Showing running time performance when performing simple effects.

Mutlishot Continuations

Unfortunately, Kotlin does not support multi-shot continuations. We cannot write effective code that performs probabilistic programming and nondeterministic path exploration. Multi-Shot delimited continuations are hard to reason with if the programmer does not have any experience with them.

Thread Safety

Currently, the library is thought to be used in single-thread contexts. The main goal of the library is not to be used in production, but to show how Kotlin would implement algebraic effects and handlers. Therefore, during the library development, we did not focus on thread safety.

Exception Safety

The library is not able to run a `finally` block within an effectful function if the computation of itself is interrupted by the handler. Let us analyze the code shown in Listing 30. When the effectful function is running, it will never print the string `"This will not be printed"`. According to Kotlin formal specification [AB], this behavior breaks the semantics of the `try …catch …finally` statement: "*If there is a finally block, it is evaluated after the evaluation of all previous try-expression block*".

Cross-Platform Compatibility

Kotlin Effects work where Kotlin coroutines do. Some target platforms are not compatible with coroutines; therefore, algebraic effect handlers will not be supported on them.

Kotlin Domain Specific Language

This minority can cause small problems when writing `handle` blocks with non-aligned `with`. Execution of an effectful function occurs due to the invocation of the `with` function operator previously explained. This problem can be avoided with a compiler warning or by implementing algebraic effect handlers directly into the compiler.

```kotlin
@Test
fun `Finally block not running`() {
    handle {
        try {
            perform(Empty)
        } catch (e: RuntimeException) {
            println("Do nothing!")
        } finally {
            println("This will not be printed!")
        }
    } with {
        throw RuntimeException()
        resume(Unit)
    }
}
```

Listing 30: Throwing an exception when resuming an effectful function. The expression within the `finally` block will not be executed.

5. Conclusions

In this thesis, we have delved into the exploration of algebraic effect handlers in the context of the Kotlin programming language.

5.1. Algebraic Effects as a Kotlin Library

Now, we can answer the question: *does it make sense to implement an algebraic effect handler in Kotlin*? The answer to this question is drawn from different perspectives. We consider only our approach taken, that is, implementing algebraic effect handlers as a library over Kotlin continuations. We have identified flaws that lead us to conclude that, while algebraic effect handlers offer significant advantages in some contexts, implementing them in Kotlin may be *not beneficial*. In the next subsections, we want to address **four main points** about our conclusions, explaining the reasons behind the given answer.

5.1.1. Effect Type Safety

One of the biggest challenges we faced during the library development is the type of verbosity for writing effectful functions. Types are essential, and they bring safeness to the code. However, the Kotlin type inference system, while being powerful, is not advanced enough to solve some type equations by itself, without giving hints to it. This verbosity can lead to code that is less *enjoyable to write and maintain*, breaking Kotlin pragmatism.

Furthermore, Kotlin, as a language designed for interoperation with Java and the support of legacy codebases, must strike a balance between introducing advanced features and maintaining backward compatibility. Therefore, it would be reasonable to tweak the type system to handle such cases. It may be interesting to integrate an effect system akin to the one used by the *Effekt* programming language, with its *contextual effect polymorphism*. The introduction of an effect system can possibly solve most of the issues with our library. Nevertheless, this is not a trivial work and it would require effort.

5.1.2. Asynchronous Programming

As explained across Chapters 2 and 3, one of the main reasons to implement *algebraic effect handlers* in a programming language is the possibility to define *advanced control flow* mechanisms, such as *lightweight asynchronous programming*. However, Kotlin already offers *coroutines* to the users, with a powerful library implementing well-known patterns, like, *sequences/generators* and *channels* [1].

5.1.3. The Need of Powerful Features

If the goal is to implement more advanced features, such as non-*deterministic path exploration*, or *probabilistic programming*, then the limitations of Kotlin's current continuation model cannot be ignored.

At the time of writing, Kotlin *does not implement* multishot continuations. A huge pool of examples seen in literature, implement algebraic effect handlers along with multishot continuations. Their absence in Kotlin did not allow us to replicate some of the results seen in other papers. The reader may point out that OCaml implements linear continuations. However, most of the examples provided by OCaml focus on lightweight concurrency, which is already implemented in Kotlin as discussed in 5.1.2.

5.1.4. Context-Receivers

Algebraic effects and handler provide great modularity within a program. Their use is easy to understand when a specific behavior must be injected in the code. *Context receivers* allow for the injection of capabilities into Kotlin code. While they do not alter the control flow in the same way that effect handlers can, there are cases where these alterations are not necessary. Let us compare briefly how can we achieve the same goal with context receivers (Listing 31) and algebraic effects with our library (Listing 32).

[1] For more information, see the `kotlinx.coroutines` library.

```
typealias IntMatrix = List<List<Int>>

interface PropertyManager {
    fun readProperty(property: String): String
}

context(PropertyManager)
fun printTable(table: IntMatrix) {
    val rows = readProperty("rows")
    val cols = readProperty("cols")
    for i in (0..rows) {
        for j in (0..cols) {
            print("${table[i][j]} ")
        }
        println()
    }
}

fun main() {
    val propertyManager = ... // From file, stdin, ...
    val table = ...
    with (propertyManager) {
        printTable(table)
    }
}
```

Listing 31: Showing how to implement a reader for properties using Context Receivers. As we can see, within the `with` block, when `printTable` invokes `readProperty` it will rely on the defined `propertyManager`.

```
typealias IntMatrix = List<List<Int>>

class ReadProperty(val property: String): Effect<String>

fun <T> (EffectfulScope<T>).printTable(table: IntMatrix) {
    val rows = perform(ReadProperty("rows"))
    val cols = perform(ReadProperty("cols"))
    for i in (0..rows) {
        for j in (0..cols) {
            print("${table[i][j]} ")
        }
        println()
    }
}

fun main() {
    val table = ...
    val handler: EffectHandler<Unit> = { effect ->
        when (effect) {
            is ReadProperty -> {
                val prop = ... // From file, stdin, ...
                resume(prop)
            }
        }
    }
    handle {
        printTable(table)
    } with handler
}
```

Listing 32: Showing how to implement a reader for properties using algebraic effect handlers. The behavior of reading a property is provided by the effect handler.

5.2. Contribution to the Language

The creation of this library has shown that the design of the Kotlin's coroutines is stable and extensible. Developers can create their own version of suspendable computation with necessary resumption mechanisms. The Kotlin Effects library proved the solidity of coroutines machinery, showing that the language is expressive enough to model other types of control flow.
The library can be freely modified by developers that want to experiment with algebraic effects and handlers.

5.3. Future Development

Having addressed what are the limitations of our approach, it would be interesting to understand how it would be possible to integrate an *effect system* in Kotlin. We think there are positive benefits, such as a better effect handling and effects composability.

To implement more powerful features, such as *non-deterministic path exploration* and *probabilistic programming*, Kotlin would need the integration of *multishot* continuations. The implementation of multishot continuations is rich in literature, *OCaml Multicore* shows a possible approach for doing so. In addition, the language would benefit from the integration of multishot continuations as it would improve the debugging experience of *JetBrains* products.

A. Kotlin Effects in Action

This appendix shows a series of Kotlin snippets using the *Effects* library. Each example is written inside a code block, explaining the meaning carefully.

A.1. Effects Test Suite

The effectful functions presented have been packed into a test suite. Most of the examples come from "*An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper*" M. Pretnar [Pre15].

A.1.1. No Effects

```
1  @Test
2  fun `No Effects`() {
3      val answer = handle {
4          42
5      } with {
6          unhandled()
7      }
8      assert(answer == 42)
9  }
```

Listing 33: The test '`No Effects`' shows what happens when no effects is performed within the scope of an effectful function. When the effectful computation ends with the return of value 42, the variable answer will be inferred of the type `Int`, making the assertion true.

A.1.2. Simple Effect

```kotlin
object Read: Effect<String>()

@Test
fun `Simple Effect`() {
    val result = handle {
        val prefix = perform { Read }
        "$prefix: hello world!"
    } with { effect ->
        when (effect) {
            is Read -> {
                resume("[info]")
            }
            else -> unhandled()
        }
    }
    assert(result == "[info]: hello world!")
}
```

.

Listing 34: This example shows how an effect of type `Read` can be used to return string values. In this case we are performing an effect of type `Read` to read a value from a source. The source, i.e., the installed effect handler, will always return the string `"[info]"`.

A.1.3. Aborting effectful function

```kotlin
object Fail: Effect<Nothing>()

@Test
fun `Abort effectful function`() {
    val result = handle {
        perform(Fail)
        0
    } with { effect ->
        when (effect) {
            is Fail -> {
                42
            }
            else -> unhandled()
        }
    }
    assert(result == 42)
}
```

Listing 35: The following example shows how the computation of an effectful function can be aborted. Performing the `Fail` effect will cause the invocation of the handler, which will return the value of 42 to the `handle`'s block call-site, making the assertion satisfied.

A.1.4. Pretnar - Simple Read Effect

```kotlin
object Read: Effect<String>()

@Test
fun `MPretnar - Simple Read Effect`() {
    handle {
        val firstName = perform(Read)
        val lastname = perform(Read)
        println("Full Name: $firstName $lastname")
    } with { effect ->
        when (effect) {
            is Read -> {
                resume("Bob")
            }
            else -> unhandled()
        }
    }
}
```

Listing 36: This example comes from *Pretnar*'s tutorial paper, where the author shows how to return a constant value when an effect handler is invoked through the performing of an effect (i.e., `Read`).

A.1.5. Pretnar - Reverse Output

```kotlin
class Print(val msg: String): Effect<Unit>

@Test
fun `MPretnar - Reverse output`() {
    val reverseHandler: EffectHandlerFunction<Unit> = {
        when (it) {
            is Print -> {
                resume(Unit)
                println(it.msg)
            }
            else -> unhandled()
        }
    }
    handle {
        perform { Print("A") }
        perform { Print("B") }
        perform { Print("C") }
    } with reverseHandler
}
```

Listing 37: The example prints the letters “A B C” in reverse order. Compared to the previous examples, we assign a handler to a variable and pass it directly to the `with the` operator. We do the same by defining the effectful function with type `EffectfulFunction<Unit>`.

A.1.6. Pretnar - Collecting Output

```kotlin
@Test
fun `MPretnar - Collecting Output`() {
    val result = handle {
        handle {
            perform { Print("A") }
            perform { Print("B") }
            perform { Print("C") }
            Pair(Unit, "")
        } with {
            unhandled()
        }
    } with { effect ->
        when (effect) {
            is Print -> {
                val (_, msg) = resume()
                Pair(Unit, "${effect.msg}${msg}")
            }
            else -> unhandled()
        }
    }
    assert(result.second == "ABC")
}
```

Listing 38: This example shows how to forward an unhandled effect to the uppermost effect handler using the `unhandled` function call.

A.1.7. State Effect - Emails Database

```kotlin
sealed class State<R> : Effect<R> {
    class Set(val key: String, val value: String): State<Unit>()
    class Get(val key: String): State<String>()
}

@Test
fun `State Effect - Emails Database`() {
    val emailDatabase = mutableMapOf(
        "bob" to "bob@tum.de",
        "alex" to "alex.12@tum.de",
    )
    handle {
        // Read existing emails
        println("Bob's email: ${perform(State.Get("bob"))}")
        println("Alex's email: ${perform(State.Get("alex"))}")
        // Create a new email address
        perform(State.Set("sofia", "sofy.best@tum.de"))
        println("Sofia's email: ${perform(State.Get("sofia"))}")
    } with {effect ->
        when (effect) {
            is State.Get -> resume(emailDatabase[effect.key])
            is State.Set -> {
                emailDatabase[effect.key] = effect.value
                resume(Unit)
            }
            else -> unhandled()
        }
    }
}
```

Listing 39: This example shows how effect handlers can be used when performing operations with a key-value database. The data is stored inside a hash-map, but changing the handler's implementation would allow reading the values from an actual database (such as Redis, mySQL, …).

B. Taming Unhandled Effects

In Chapter 4 we saw what the main limitations of our library were. Compared to the effects implemented in OCaml, our library does not use any *effect system*, but relies on Kotlin's type system, which cannot guarantee some properties. Moreover, the introduction of an effect system into the compiler *could possibly break legacy programs*. To overcome these drawbacks, it is possible to create a Kotlin *compiler plugin*, to be used while the effect library is enabled. There are several ways to implement this tool. The one we will present is based on a currently ongoing *research project* at JetBrains: the *Kotlin Verification Framework*, implemented as a compiler plugin. The tool is presented at high level, and we do not provide any implementation.

The framework, at the time of writing, is used to verify Kotlin contracts. Without delving into much detail, Kotlin allows developers to specify contracts on functions, but those are not verified, but trusted by the compiler. Using the framework, it is possible to verify if these contracts are valid with respect to the function's body, or not. We can use the plugin to verify some properties on effectful functions and the existence of a handler able to manage them. This chapter will expose how this compiler plugin could be exploited to create a tool that mimics an effect system.

B.1. The Viper Infrastructure

The Verification Framework compiler plugin uses a verification infrastructure developed by the *Programming Methodology Group* at Eidgenössische Technische Hochschule Zürich (ETH Zurich). **Viper** is an *intermediate verification language* for verifying programs [MSS16]. The language is used to verify programs that access heap, through the *separation-logic*, *an extension of Hoare's logic that reason about memory accesses* [Rey02].

A Viper program is a sequence of method declarations containing a contract: a set of *pre-conditions* and *post-conditions* that the method must hold when used.

B.2. Kotlin to Viper

It is possible to transpile Kotlin in Viper. The verification framework takes a Kotlin function, and it transforms it into a Viper method. For being able to verify effectful functions and the existence of handlers, we need to define what are effects into Viper, along with a set of axioms on them and how they interact with algebraic effect handlers. Therefore, the next subsections will focus on how the definition of a domain for effects can be achieved and how effect handlers are represented.

B.2.1. Encoding Algebraic Effects and Handlers

As stated in Viper's documentation [1], a *domain* allows the definition of additional types, mathematical functions and axiomes that provide their property.

We start by showing how we decided to encode the effects. In the Listing 40 we define a new domain called `Effect`. In Viper, it is not possible to define elements of a domain in an explicit way; we have to create a value's constructor. That is, given an effect in Kotlin (so classes or objects inheriting the `kotlin.effects.Effect` interface), create a new function that instantiates an instance of that effect. Each function within the domain is marked as unique, guaranteeing that the returned effect invoked by a function is uniquely determined. This will be useful when defining axioms for handlers (see next paragraph).

```
1 domain Effect {
2     unique function Fail(): Effect
3     unique function YieldInt(): Effect
4     unique function Read(): Effect
5 }
```

Listing 40: The domain `Effect` encodes all the effects defined within a Kotlin program. Each effect is represented by a *unique* function, returning an instance of the given effect.

For effect handlers, we introduce a new domain called `Handle` (refer to Listing 41). Within this domain, we provide functions for defining new handlers when given a set of effects (as demonstrated by `NewHandler`), as well as a function for obtaining the set of effects that a particular handler can handle (illustrated by `GetEffects`). Given two handlers $h_1, h_2 \in$ Handler, the function `MergeHandlers`, creates a new handler capable of handling both effects in h_1 and h_2.

[1] Viper's documentation can be found on the following link: https://viper.ethz.ch/tutorial/.

```
domain Handler {
    /* Constructs an Handler that handles the given effects. */
    function NewHandler(effects: Set[Effect]): Handler
    function GetEffects(h: Handler): Set[Effect]
    /* New constructor for merging handlers. */
    function MergeHandlers(h1: Handler, h2: Handler): Handler

     axiom ax_GetEffects {
        forall s: Set[Effect] ::
            exists h: Handler :: {NewHandler(s), GetEffects(h)}
                h == NewHandler(s) && GetEffects(h) == s
    }

    axiom ax_AllDiff {
        forall h1: Handler, h2: Handler :: { GetEffects(h1),
        ↪ GetEffects(h2) }
            GetEffects(h1) != GetEffects(h2) ==> h1 != h2
    }

    axiom ax_MergeHandlers {
        forall h1: Handler, h2: Handler, h3: Handler :: {GetEffects(h1),
        ↪ GetEffects(h2), GetEffects(h3)}
            h3 == MergeHandlers(h1, h2) ==> GetEffects(h3) ==
            ↪ (GetEffects(h1) union GetEffects(h2))
    }
}
```

Listing 41: The encoding for Algebraic Effect Handlers. The `Handler` domain. A new value from the handler domain is constructed using the `NewHandler` function, given a set of algebraic effects.

The handler domain defines three important axioms, explained as follows.

Axiom 1 (Get Effects from Handler) *For all the set of effects $S \in \mathcal{P}(Effect)$, there exists a handler $h \in Handler$, constructed by the* `NewHandler` *function, such that:*

$$GetEffects(NewHandler(S)) = S$$

Axiom 2 (Handlers Inequality) *Given two effect handlers, $h_1, h_2 \in Handler$, we say*

that they are different if the following statement holds true:

$$h_1 \neq h_2 \equiv GetEffects(h_1) \neq GetEffects(h_2)$$

Axiom 3 (Merge of Handlers) *Given two effect handlers $h_1, h_2 \in Handler$, we can say that $h_3 \in Handler$ is the **merge** between $h_1 \cup h_2$ if:*

$$\forall h_1, h_2, h_3 \in Handler. h_3 = h_1 \cup h_2 \implies GetEffects(h_1) \cup GetEffects(h_2) = GetEffects(h_3)$$

At last, we need to define two predicates to define if given an effect $e \in$ Effect it is handled by an handler $h \in$ H.

Predicate 1 *We say that an effect $e \in Effect$ is **handled** by an handler $h \in Handler$ if:*

$$e \in GetEffects(h)$$

Likewise, we can say that a set of effects $S \in \mathcal{P}(\text{Effect})$ is handled by a given handler h if $S \subseteq \text{GetEffects}(h)$.

B.2.2. Transforming Effectful Functions

We must determine how to represent an effectful function in Viper techniques once we have the domains for effects and handlers. One possible solution to do so is to collect all effects performed by an effective function ($\text{E} \subseteq \mathcal{P}(\text{Effect})$) and create a Viper method taking a handler h as parameter, requiring as a precondition that the predicate `EffectsAreHandled` holds, with the given handler h and set of effects E.

```
1 method EffectfulFunction(h: Handler)
2     requires EffectsAreHandled(h, E)
```

Listing 42: Encoding of effectful functions. As explained in the respective paragraph, the set `E` contains all the effects triggered in the effectful function considered.

Thus, a Kotlin's effectful function will be transformed as shown in the Figure B.1. When an effectful function is invoked, the corresponding handler within the `with` block will be used to create a new Viper handler.

```
handle h1@{
    // ...
    perform(Yield(20))
    // ...
    perform(Fail)
} with { effect ->
    when (effect) {
        is Fail -> ...
        is Yield -> ...
    }
}
```

```
method H1(h: Handler)
    requires EffectsAreHandled(h,
    ↪ Set(Effect_Yield(),
    ↪ Effect_Fail()))

method Driver() {
    var e := Set(Effect_Yield(),
    ↪ Effect_Fail())
    var h := NewHandler(e)
    H1(h)
}
```

Figure B.1.: Converting a Kotlin effectful function into Viper code. Executing Viper on this program will not generate any warnings, as the precondition for the `H` method is satisfied. On the left side, we have the Kotlin code converted into an equivalent in Viper (right side). If we had to remove the effect `Yield` or `Fail` from `e`, Viper will warn us with the following message: `The precondition of method H1 might not hold`.

B.2.3. Limitations of the Analysis

Although this analysis is valuable, it is not a silver bullet and cannot replace an *effect system*. The verification process we conducted is sound, but not exhaustive, which means that it may produce false positives. This issue becomes relevant when the program employs a more structured control flow, since the analysis is not *path-sensitive*.

Effects with Type Parameters

Additionally, this encoding does not support effects containing *type parameters*. Therefore, we have to monomorphize each generic effect into a new construct, that is, `Yield[A]` will be instantiated as `Yield_A` and so on.

List of Figures

List of Listings

Bibliography

[AB] M. Akhin and M. Belyaev. “Kotlin language specification.” en. In: ().

[Bie+19] D. Biernacki, M. Piróg, P. Polesiuk, and F. Sieczkowski. “Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers.” In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371116.

[Bin+18] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. *Pyro: Deep Universal Probabilistic Programming*. 2018. arXiv: 1810.09538 [cs.LG].

[BSO18] J. I. Brachthäuser, P. Schuster, and K. Ostermann. “Effect Handlers for the Masses.” In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276481.

[BSO20] J. I. Brachthäuser, P. Schuster, and K. Ostermann. “Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism.” In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428194.

[BWD96] C. Bruggeman, O. Waddell, and R. K. Dybvig. “Representing Control in the Presence of One-Shot Continuations.” In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI ’96. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 99–107. ISBN: 0897917952. DOI: 10.1145/231379.231395.

[Com] U. Computing. *The Unison language*. https://www.unison-lang.org/. (Accessed on 08/20/2023).

[Dan94] O. Danvy. “Back to direct style.” In: *Science of Computer Programming* 22.3 (1994), pp. 183–195. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(94)00003-4.

[Eli+21] R. Elizarov, M. Belyaev, M. Akhin, and I. Usmanov. “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! 2021. Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 68–84. ISBN: 9781450391108. DOI: 10.1145/3486607.3486751.

[Fou] K. Foundation. *KEEP/proposals/context-receivers.md at master · Kotlin/KEEP · GitHub*. https://github.com/Kotlin/KEEP/blob/master/proposals/context-receivers.md. (Accessed on 09/12/2023).

[Git] GitHub. *GitHub - github/semantic: Parsing, analyzing, and comparing source code across many languages*. https://github.com/github/semantic. (Accessed on 09/03/2023).

[Hil15] D. Hillerström. "Handlers for Algebraic Effects in Links." en. In: (2015).

[HL18] D. Hillerström and S. Lindley. "Shallow Effect Handlers." In: *Asian Symposium on Programming Languages and Systems*. 2018.

[Kis] O. Kiselyov. *Continuations and Delimited Control*.

[Lei14] D. Leijen. "Koka: Programming with Row Polymorphic Effect Types." In: *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), pp. 100–126. DOI: 10.4204/eptcs.153.8.

[Lin14] S. Lindley. "Algebraic Effects and Effect Handlers for Idioms and Arrows." In: *Proceedings of the 10th ACM SIGPLAN Workshop on Generic Programming*. WGP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 47–58. ISBN: 9781450330428. DOI: 10.1145/2633628.2633636.

[LMM17] S. Lindley, C. McBride, and C. McLaughlin. *Do be do be do*. 2017. arXiv: 1611.09259 [cs.PL].

[Met] Meta. *React*. https://react.dev/. (Accessed on 09/03/2023).

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning." In: *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*. VMCAI 2016. St. Petersburg, FL, USA: Springer-Verlag, 2016, pp. 41–62. ISBN: 9783662491218. DOI: 10.1007/978-3-662-49122-5_2.

[OCa] OCaml. *Multi-shot continuations gone forever? - Ecosystem - OCaml*. https://discuss.ocaml.org/t/multi-shot-continuations-gone-forever/9072/2. (Accessed on 08/26/2023).

[Pap] G. Pappalardo. *Gabryon99/Kotlin-effects: A Kotlin library implementing algebraic effect handlers. https://github.com/gabryon99/kotlin-effects*.

[PP09] G. Plotkin and M. Pretnar. "Handlers of Algebraic Effects." In: *Programming Languages and Systems*. Ed. by G. Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. ISBN: 978-3-642-00590-9.

[Pre15] M. Pretnar. "An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper." In: *Electron. Notes Theor. Comput. Sci.* 319.C (Dec. 2015), pp. 19–35. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2015.12.003.

[Rec] I. N. de Recherche en Informatique et en Automatique. *OCaml - Chapter 12 - Language extensions*. https://v2.ocaml.org/manual/effects.html. (Accessed on 08/26/2023).

[Rey02] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0769514839.

[Siv+21] K. C. Sivaramakrishnan, S. Dolan, L. White, T. Kelly, S. Jaffer, and A. Madhavapeddy. "Retrofitting Effect Handlers onto OCaml." In: *CoRR* abs/2104.00250 (2021). arXiv: 2104.00250.

[TG08] F. A. Turbak and D. K. Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008. ISBN: 0262201755.

[Uni] Unison. *Abilities and ability handlers · Unison programming language*. https://www.unison-lang.org/learn/language-reference/abilities-and-ability-handlers/. (Accessed on 09/06/2023).

[VP21] P. E. de Vilhena and F. Pottier. "A Separation Logic for Effect Handlers." In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434314.