The modern landscape of managing effects for the working programmer

Master of Science in Technology
Thesis
University of Turku
Department of Computing
Software Engineering
2023
Jaakko Paju

Supervisors:
Jaakko Järvi

UNIVERSITY OF TURKU
Department of Computing

Jaakko Paju: The modern landscape of managing effects for the working programmer

Master of Science in Technology Thesis, 109 p., 2 app. p.
Software Engineering
May 2023

The management of side effects is a crucial aspect of modern programming, especially in concurrent and distributed systems. This thesis analyses different approaches for managing side effects in programming languages, specifically focusing on unrestricted side effects, monads, and algebraic effects and handlers. Unrestricted side effects, used in mainstream imperative programming languages, can make programs difficult to reason about. Monads offer a solution to this problem by describing side effects in a composable and referentially transparent way but many find them cumbersome to use. Algebraic effects and handlers can address some of the shortcomings of monads by providing a way to model effects in more modular and flexible way. The thesis discusses the advantages and disadvantages of each of these approaches and compares them based on factors such as expressiveness, safety, and constraints they place on how programs must be implemented. The thesis focuses on ZIO, a Scala library for concurrent and asynchronous programming, which revolves around a ZIO monad with three type parameters. With those three parameters ZIO can encode the majority of practically useful effects in a single monad. ZIO takes inspiration from algebraic effects, combining them with monadic effects. The library provides a range of features, such as declarative concurrency, error handling, and resource management. The thesis presents examples of using ZIO to manage side effects in practical scenarios, highlighting its strengths over other approaches. The applicability of ZIO is evaluated by implementing a server side application using ZIO, and analyzing observations from the development process.

Keywords: functional programming, side effect, algebraic effect, monad, Scala, ZIO

Contents

1 Introduction

Modern programs interact with their environment, such as users, files, databases, message buses, and/or other applications. They should be able to serve hundreds, thousands, or sometimes even millions of users simultaneously, utilizing the underlying hardware efficiently. They are expected to be available and working every day of the year, around the clock. These programs are expected to be robust and resilient, meaning they should react to failures in a predictable and well-defined manner. At the same time, programs should be fast to develop and modify when adding new features or changing existing ones, i.e., applications are expected to be modular.

To write programs that satisfy these demands is no easy task for a programmer. Many of the key tenets of this task is managing side effects and concurrency, and exceptions arising from those. Side effects, also known as computational effects or just effects, are a byproduct of calling a function that causes or observes changes in its environment. Concurrency is the ability to interleave several units of work to be executed simultaneously.

Modular and expressive management of side effects, errors and concurrency is something that current, imperative, mainstream languages do not excel at. The academic research community has, however, introduced several techniques that make it possible to work with effects in a compositional and expressive way. These more sophisticated methods for managing effects are based on functional programming. Even though the theoretical foundations of functional programming date back to

almost a hundred years, when lambda calculus was invented in the 1930s, functional programming languages have not became mainstream. All of today's most widely used programming languages, as ranked by TIOBE Index [1] and Stack Overflow [2], are fundamentally imperative.

Many functional concepts, however, have been recognized to be valuable in modern software development. Functional programming promises ease of reasoning about program behavior – immutability gives referential transparency and consequently equational reasoning, and composability. These functional concepts hold promise for handling effects, concurrency and modeling complex business logic, which are the core of many modern applications. Features like immutability, lambdas and higher-order functions, have found their way to imperative and object-oriented mainstream languages like JavaScript, Python, Java, and C#.

It is not too much of a stretch to say that functional features are currently disrupting the field of programming. The purpose of this thesis is to analyze and understand how these features can be utilized. The aim is to bridge the gap between solutions studied and used in academia and the dominating technologies used in the industry by studying different methods of managing effects from a practical perspective.

The thesis studies three different approaches to side effects; unrestricted side effects, monads, and algebraic effects and handlers, which are analyzed in the context of Koka and Unison programming languages. The thesis also provides a brief overview of a bleeding-edge approach to managing effects, called capability-based effects. As a vehicle for concrete experimentation, we use and study a Scala library called ZIO, which can support the above programming approaches. The approaches are studied in terms of how they affect the implementation of programs. The thesis seeks answers to the following research questions about different approaches to managing effects:

RQ1: How expressive and compositional the approach is?

RQ2: What are the safety guarantees the approach offers?

RQ3: Does the approach place limitations on how programs can be written?

These are broad questions and it is worth mentioning that a reader who is looking for an unambiguous, concise and one-size-fits-all answer to these questions might be disappointed by the subject's multifaceted nature. Instead of easy answers, the thesis seeks to provide a comprehensive understanding of the domain of computational effects, and offers analysis of the main approaches for managing effects from different perspectives.

Beyond the issues addressed by these research questions, many other factors have an impact on whether the programming techniques of monads and algebraic effects will gain wider adoption or not. For example, testability of monadic code or code built with algebraic effects would be an interesting area of research. Sociological aspects, programmers' perception of complexity etc. are also certain to have an effect on adoption of these advanced techniques. This thesis discusses a few observations about testing and programmers' preferences regarding programming languages in the case study part but otherwise shies away from these topics.

Chapter 2 studies the definition of effects and introduces several common types of effects. The chapter also includes a discussion of concurrency and its implementation approaches. Also, Scala and its relevant features are introduced in this chapter. Chapter 3 gives an introductory account on how effects are included in programming languages, and how they can be managed. Chapter 4 introduces ZIO and explores how it approaches effect management. Chapter 5 presents a case study that evaluates the practical applicability of ZIO for server side application development. The last chapter compares the properties of different methods for managing side effects and draws conclusions from them.

2 Background

Functional programming uses immutable values and mathematical functions, also known as pure functions, to build programs. Similarly to imperative procedures, pure functions take parameters as input and compute some output. Unlike imperative procedures, however, pure functions are **only** allowed compute output value based on their input and cannot have any other observable effects. Given the same inputs, a pure function must always evaluate to the same outputs. Abstraction and reuse, similarly to in imperative programs, are achieved by composing functions by passing the output of the previous function to the next function's input. The entire program can be seen as a large function composition of all functions used in the program.

A major difference between imperative and functional programming is in how one can reason about procedure or function compositions. Any expression in functional programming can always be *substituted* with its value without changing the meaning of the program. The same does not apply in imperative programming. There is an implicit temporal coupling between imperative statements, since a statement may depend on the state set by previous statements. Because of this, reordering procedure calls or substituting any procedure call with its return value might change the meaning of the program. [3, Chapter 1]

A program is considered to be *referentially transparent* if it is possible to substitute an arbitrary expression in the program with its corresponding value without

changing the meaning of the program in any way. Referentially transparent programs are easier to understand since they enable *equational reasoning*, also known as *local reasoning*. When composing pure functions, one does not have to understand their implementation, because the only effect the function is allowed to have is to return a result. A developer can only focus on the *function's signature* and its specification, that is, what are the inputs and what is the output. Compilers can also take advantage of referential transparency by safely reordering expressions, evaluating expressions at compile time, memoizing results or by completely skipping the evaluation of expressions that are not required.

Referential transparency is one of the biggest differentiating factors between functional and imperative programming. Abandoning referential transparency has wide-reaching implications. In practice, it makes it much more difficult to refactor and develop programs. Developers are required to be more disciplined and to have wider knowledge of the whole program in order to not unintentionally cause defects. This is particularly evident when programming in the presence of concurrency, where side-effects can lead to race conditions and hard-to-reproduce errors. [3, Chapter 3]

This chapter introduces first what effects are and discusses certain common effect types in more detail. It then describes what concurrency is, how it can be achieved and what kind of problems it causes. Structured concurrency, a concept for defining semantics on how concurrent workflows interact, is also introduced. Lastly, the history and features relevant to managing side effects of the Scala programming language are introduced.

2.1 Effects

Constructing programs only by composing pure expressions without any notion of impurity is quite limiting, to say the least. To be useful in practice, programs depend on effects. An expression is said to have an effect if its sole purpose is not to evaluate

to a value and its evaluation requires interacting with the outside world. Printing to the console, accessing the system clock or doing IO are all examples of effects. There is no unambiguous and exact definition of what an effect is, although the concept has been given, somewhat differing, characterizations by many.

Cartwright and Felleisen [4] suggest that "A complete program is thought of as an agent that interacts with the outside world, e.g., a file system, and that affects global resources, e.g., the store [mutable memory]". They continue by stating that every phrase in a program could be classified to either a value or an effect. A value is a referentially transparent expression, while an effect is an interaction with resources allocated for the program. When an effect is encountered, the control is transferred to a "central authority". The central authority manages the use of all resources the program has access to. They continue to describe the interaction between an effect and the central authority:

> An effect is most easily understood as an interaction between a subexpression and a central authority that administers the global resources of a program. (..) Given an administrator, an effect can be viewed as a message to the central authority plus enough information to resume the suspended calculation.

Peyton Jones and Wadler [5], as well as Lindley, McBride, and McLaughlin [6] see the distinction between expressions and effects as *being* vs. *doing*. This observation is quite interesting since it brings up the concept of computations as values. Certain approaches deliberately differentiate computations from values, while some deliberately unify them. This thesis discusses how separation of effects from values applies to monadic effects and algebraic effects with handlers, together with the concept of a central authority presented earlier.

Different effects could be categorized as *internal* or *external*. Unlike internal effects, external effects can be observed from the outside. In the context of a whole

program, the only external effect is IO, while other effects are internal. In the context of a function, matters are more complicated, since effects such as mutable state, raising exceptions, and concurrency can be both internal or external, depending on the specific situation.

2.1.1 Mutability

Mutability means that a program is able to change its state, usually by mutating data stored in some memory location, and that it is possible to detect a state change by observing the changed value. Several control structures and language features require mutability. The destructive assignment operation found in almost every mainstream language is by definition mutation [3, Chapter 3]. Looping constructs such as `for-` and `while` loops, and iterators found in many standard libraries, rely heavily on the notion of mutation. Also parts of some well known algorithms, like the swap operation in quicksort, can be expressed trivially as mutation.

In practice, almost all programs have some state that determines how the program reacts to input. Real-world examples of state include the location of characters in a game, registered users in an application and cursor position in the buffer when reading bytes from a socket. In the presence of concurrency, when parallel computations are expected to interact with each other, mutability in one form or another is needed to indicate if a computation is still on-going, completed or has encountered an error.

2.1.2 Exceptions

Another very common effect is the ability to signal about exceptional conditions where the program is unable to compute a result or execute a command. This signaling is achieved by raising an error or exception. An exception could contain information about the condition that caused it, for example malformatted input, and

that could possibly be later used when *handling* or recovering from the exception. There are several common reasons why exceptions arise. They usually fall into two categories: technical or logical. [7]

Logical exceptions are usually caused by failing to meet some preconditions regarding the program's state or a function's parameters. A function may have assumptions about its inputs — a string may need to be in a format that matches a schema in order to parse it successfully, or an integer may need to be positive and less than a certain threshold to represent a year. Sometimes inputs must be compatible with other inputs. An example of this is accessing an array by its index where the accessed index must be less than or equal to the size of the array, or attempting to access authorized content before proper authentication and authorization process.

Technical exceptions are usually related to IO, external events, the runtime environment, or the programming language itself. They can further be divided into synchronous and asynchronous exceptions. Peyton Jones describes synchronous exceptions as something that "arise as a direct result of some piece of code" [8]. On the other hand, asynchronous exceptions are caused by external events and they cannot always be tied to the execution of a particular line of code. In some way, logical and synchronous exceptions are expected exceptions, and asynchronous exceptions are unexpected.

Many synchronous exceptions are related to IO. If attempting to interact with a file that does not exist or the current permissions are not sufficient, the result will likely be an exception of some sort. A significant source of exceptions is communicating over the network with a remote party. Everything from name resolution, routing, transport protocol or communication schema could go wrong. A remote component in a distributed system could be completely unavailable due to a network error or an internal error in that specific component. IO problems also arise when trying to perform an action before initialization, for example via a database

connection, file descriptor or IO port.

Other synchronous exceptions may be caused by division by zero or a non-exhaustive pattern match. Probably the most well known synchronous exception is the infamous null reference error, where the program is trying to dereference a pointer that does not point to a valid memory location. In languages that support direct memory access, an attempt to access memory outside of the allowed memory range leads to a program or operating system level exception. [8]

Asynchronous exceptions usually originate from the runtime environment of the program, operating system, concurrency, or user interruption. Asynchronously raised exceptions are characterized by the fact that they could occur at an arbitrary point in time [9]. An example of this is a situation where a thread interrupts the execution of another thread. The whole program could also be interrupted by a user (for example by pressing Ctrl+C) or the runtime, possibly due to a critical error in the program or operating system. Resource exhaustion is another common cause of asynchronous exceptions. Errors like stack overflow or out of memory can happen every time new memory is required from the stack or heap, thus those are categorized as asynchronous. Many environments also support dependencies to libraries that are loaded/linked dynamically at run time. The programmer cannot always specify the exact time when dynamic loading should take place, and for this reason failing to load required dependencies could be considered an asynchronous exception. [8]

Exceptions can also encode another related and important concept, *optionality*. Encoding optionality via exceptions is achieved by raising an exception that contains only a value of the unit type[1], signaling that no result could be computed and there is no additional information about the exception. Optionality is an approriate choice instead of exceptions when the cause of the exception is trivial. Such cases

[1]A type whose cardinality is 1 (i.e., that has only one value) and thus does not contain any information.

include unsuccessfully querying a row from a database with a specific id, searching an element from an array or trying to find a substring from a string.

Usually, the semantics of raising and handling an error are to interrupt the normal control flow of the program and transfer the execution to the closest appropriate *exception handler*. An exception handler decides if and how to continue the execution, or whether to let the exception bubble up the layers of exception handlers. This "short-circuiting" semantics is a natural way to think and program in the presence of errors. However, the ability to raise errors from an arbitrary location can make it difficult to understand the meaning of the program and prove its correctness. It is a challenge to ensure that all exceptions that may be raised are handled appropriately. Lazy evaluation complicates things even further. The evaluation order in a lazily evaluated language may not be obvious to the programmer. This makes it harder to define clear semantics for exceptions. [7]

Effective and thorough exception handling is one of the most important practices in successful software engineering. Conversely, the inability to do so is one of the most significant factors that causes bugs and failures in software systems. A 2014 study [10] by researchers of the University of Toronto studied multiple popular open source distributed software systems, such as Redis, Hadoop and Cassandra and found that a large portion (35%) of catastrophical failures were caused by trivial mistakes in error handling code. Such mistakes include practices like omitting error handling code completely and writing a TODO-comment instead. In addition to failures, inadequate error handling may expose security vulnerabilities in the system.

2.1.3 IO

Programs need the ability to interact with the external world, i.e., with a user, other programs, or devices and sensors. IO is the medium to carry out these interactions. Like interaction in general, IO is often bidirectional — the term IO is a shorthand for

input and output. Input is the ability to observe changes and to receive information from other parties, output enables a program to cause changes in the environment and to dispatch information to others.

Many IO effects are about interacting with the user. Probably the most well-known and fundamental form of user interaction is to display text and graphics by changing pixels on the screen. Another common type of user interaction is via the console, which consists of printing characters to standard output and reading user input from standard input. The use of external devices such as playing sounds from speakers, recording sound from a microphone, or receiving user input from the keyboard, mouse, and touchscreen, is essential in user interaction.

In addition to user interaction, a program can also use devices for other purposes. For example, reading the time from the system clock, requesting the current temperature from a sensor, or setting a digital output to 1 or 0.

Often programs need the ability to store data that persists even when the program is restarted. This is achieved by using a device that allows reading from and writing to a non-volatile memory, such as a hard drive or memory card. Usually an operating system abstracts this persistent data store by providing a file system. However, many embedded devices still communicate directly with persistent memory devices.

The reason for a program to exist is to eventually have an effect on the surrounding world. As IO is the only way to achieve this, it fundamentally distinguishes IO from other effects. Where other effects might be useful for structuring computations and expressing computations in certain ways, IO is *the reason* for programs to exist in the first place [8]. To put it the other way around, it would be impossible to detect if a program is running or not if it would not be interacting with its environment.

2.2 Concurrency

Computer programs should be able to run multiple workflows interleaved (concurrency) or at the same time (parallelism). Programs often have many simultaneous users, all of whom should be able to use these programs independently of each other. Also, it is characteristic of IO that a large portion of time is spent waiting for a response, rather than calculating results with local computing resources, mainly CPUs. When several operations can be performed in parallel or while waiting for IO responses, performance improves and the underlying hardware is utilized efficiently.

2.2.1 Concurrency adds complexity

Often workflows must interact with other concurrent workflows. A parent workflow might spawn multiple child workflows and split a task between them. In some situations, one might run several workflows in parallel and choose the result that is computed the fastest, discarding all other results. Concurrent workflows sometimes must use a shared resource, like mutable memory, file, or database connection.

At first glance, these interactions may seem not particularly problematic, but concurrency complicates programs significantly. By default the execution order of concurrent workflows is nondeterministic because of how tasks are scheduled (usually by the operating system) to run on actual hardware. Many statements in a programming language are compiled to or interpreted as several CPU instructions, executed sequentially at different clock cycles. A canonical example is the increment operator (`++` or `+=`) that first reads a variable's value with one instruction and then sets it to a new value with another. If another parallel workflow is updating the same variable at the same time, it might see the value between the two instructions, even though that is rarely the desired behavior. These kinds of situations are called *race conditions*.

In order to prevent race conditions, explicit countermeasures are required. One

such countermeasure is *synchronizing* accesses to shared resources. Usually this means that before a workflow can enter a particular section of a program or use a shared resource, it must acquire an exclusive *lock*. This means that only one of the workflows has access to the resource at a given point of time. Another countermeasure, applicable to shared memory, is using atomic compare-and-swap operations [11], which enable updating some data and succeeding only if the data was not modified by another workflow during the update.

Locks and atomic references are examples of low-level tools for managing concurrency. Each tool comes with a set of trade-offs. When a workflow must acquire multiple locks, a possibility for *deadlock* arises. Deadlock is a situation where two concurrent workflows are blocked by each other and neither can continue until the other releases a lock they are holding. An atomic operation can fail, and the failure must be handled, for example, by retrying until the operation succeeds. Basic atomic operations usually cannot guarantee the atomicity of operations spanning over many atomic references.

2.2.2 Concurrency primitives

In practice concurrency can be implemented with many different constructs. The lowest-level construct commonly accessible to programming languages is a *thread*. It is an OS level abstraction for concurrent execution. Each thread has its own stack, instruction pointer and CPU register values. All threads created by a single process share the same memory space, i.e., they are able to read from and write to the same shared memory blocks.

Threads are run on the actual hardware of the computer; a modern multi-core CPU can execute several threads in parallel. The OS *schedules* different threads for execution, and after a thread has been executing for a scheduled amount of time, the operating system interrupts the execution and switches the execution to a different

thread. The operation where a CPU core switches the execution from one thread to another is called *context switch*. Context switch requires that CPU registers and stack of the previous thread are saved, and respectively registers and stack of the new thread is loaded to the CPU.

Traditionally a thread has been the concurrency primitive to turn to when some form of parallelism is required. Threads, however, are not a lightweight construct and they can exist only in limited numbers, usually in the thousands. Context switching between threads involves a significant amount of work, and thus causes performance overhead. Often a context switch defeats many optimizations in contemporary CPUs like caching, pipelining and speculative execution, which in turn amplifies the performance overhead. The issue manifests itself particularly in highly concurrent scenarios where computations are IO bound, which is usually the case in web and enterprise applications.

A common way to constrain the total number of threads and increase their reuse is to collect multiple threads into a *pool* of threads, where tasks could be submitted for execution instead of operating with individual threads. Once a task is submitted to a thread pool, it is queued and run once there are available threads. This way many concurrent workflows could be *multiplexed* into a smaller number of physical threads. When fewer threads are created and reused by multiple concurrent workflows, the intent is to decrease the number of context switches in order to achieve performance gains.

Thread pools do not solve the problem of blocking threads while they are waiting for an IO operation or another thread to complete. When a thread blocks, the OS puts it in a waiting state, meaning that its execution is not continued until the event it is waiting on is triggered. The ideal solution would be that no physical threads are blocked, and blocking is only semantic. This is not possible when threads are preemptively scheduled by the OS. A solution is to change the scheduling model from

preemptive to *cooperative*. Cooperative scheduling means that when a workflow is about to be blocked, it will register itself to be scheduled once all of its dependencies are met, and yield the control to other workflows. In this model no physical threads need to be blocked.

Cooperative scheduling is usually implemented by a runtime environment, programming language, or library that runs on top of preemptively scheduled OS threads. Event loop is a common pattern for implementing cooperative scheduling, and it is used extensively in asynchronous IO or single-threaded environments like JavaScript. The idea is to have a queue that contains computations waiting to be executed, and an event loop that picks up and executes tasks from the queue one at a time. Once a task is about to do a blocking operation, it registers a callback. The callback is invoked when the blocking operation completes, and it will add another task to the queue that represents the remaining of the workflow.

Another way to implement cooperative scheduling is *fibers*. They are lightweight threads that are managed and scheduled in the application instead of OS. Each fiber contains a stack and possibly error handlers or thread-local variables, similar to a thread. Fibers require a scheduler that determines what fibers to execute on actual OS threads. The scheduler can multiplex many fibers to run on smaller number of physical threads. Fibers could exist in the hundreds of thousands or even millions; switching execution from one fiber to another is very cheap in comparison to a context switch between threads. The fiber scheduler can assign a fiber to execute on a specific CPU core, which will make it easier to reap benefits from CPU optimizations like caches.

2.2.3 Structured concurrency

When parent workflow spawns many child workflows, it is common that if one of the children encounters an error, the result cannot be computed at all and thus the

results of other sibling workflows are not needed anymore. A similar situation may occur with racing workflows: when the first workflow successfully computes a result, the results of other workflows are no longer needed. In both of these situations it would be ideal to cancel the workflows whose results are not needed, to preserve compute resources and make sure that no concurrent workflows remain in execution. Traditional concurrency primitives, such as threads, do not offer this kind of control out of the box.

A solution to this is *structured concurrency* [12], [13], which makes it possible to define clear semantics on if and how a child workflow could outlive its parent. The basic idea of structured concurrency is that there is a way to govern how child workflows are handled when the parent workflow completes (by succeeding or failing), or when there is an error encountered in any of the sibling workflows. For example, for a parent workflow that spawns child workflows one would like to define whether the children should be awaited, cancelled, or left orphaned when the parent completes or is cancelled before the children are finished.

Native support for structured concurrency in programming languages is still quite rare, but it has been added to programming languages at an accelerating pace. Kotlin added structured concurrency in 2018 [14], Swift 5.5 in 2021 [15] and Java 19 in 2022 [16]. The feature will probably find its way into more programming languages in the future.

It is difficult to write correct and concurrent programs. Knowledge of concurrency primitives and tools, such as different data structures and CPU instructions, and possible concurrency hazards are required. One has to be especially careful about race conditions, which are not always obvious. Ideally, concurrency could be implemented with high-level code, using operations that take into consideration possible concurrency issues, and deal with low-level details.

2.3 Scala

Scala [17] is a high level, statically typed, compiled, and garbage collected programming language, that is both functional and object-oriented. It is eagerly evaluated by default, but it also supports lazy evaluation. The first release was in 2004 and the latest version is 3, which was released in 2021. Version 3 is exclusively used in this thesis. The initial and current lead designer of the language is Martin Odersky, a professor at the École polytechnique fédérale de Lausanne. Scala's roots are thus in academia, but its approach is pragmatic.

Scala is most commonly run on the Java Virtual Machine (JVM), but also JavaScript and native code are supported compilation targets. When running on the JVM it is possible to use Java code directly from Scala. The Scala standard library even contains functions for converting Java data types to their Scala counterparts. This gives access to a huge number of Java libraries.

Scala aims to blend the Functional programming (FP) and Object-oriented programming (OOP) paradigms and as a result has features from both. Many OOP concepts like classes, objects, interfaces and subtype polymorphism are supported. In fact, every value in Scala is an object. Scala uses class-based objects with attributes and methods, and supports multiple inheritance. Scala supports generics with lower and upper subtype constraints as well as declaration-site variance. The language also includes many imperative constructs, like loops and mutable variables, that are commonly used in other OO-languages. What is perhaps less common in most OO-languages is that in Scala everything is an expression, including control structures like `if/else`, `try/catch`, and loops. Listing 1 demonstrates the basic syntax of Scala.

Due to Scala's object-oriented nature, every object is part of a type hierarchy. On top of the hierarchy is `Any`, which is the supertype of all other types. Below `Any`

Listing 1 Basic syntax of Scala.

```
trait Foo // Define an interface
class Bar extends Foo // Define a class inheriting from Foo

// Define variables/constants
var mutableFoo: Foo = Bar() // Explicit type is Foo
val immutableBar    = Bar() // Inferred type is Bar
lazy val lazyPlus   = 1 + 1 // Computed lazily and cached

// Type argument here is Int
val genericType: List[Int] = List(1, 2, 3)

// Type parameters are declared between '[' and ']'
def genericMethod[A](a: A): A = a

// Type parameter constraints:
// 'A' must be supertype of 'Bar' and  'B' must be subtype of 'Foo'
def typeBounds[A >: Bar, B <: Foo](a: A): B = ???

// ??? is defined in the standard library. It can replace any
// expression; it's type is Nothing, the bottom type
def `???` : Nothing = throw new NotImplementedError

// => specifies 'by-name' calling convention:
// The parameter n is evaluated every time it is used (2 times here)
def byNameParameter(n: => Int) = n + n
```

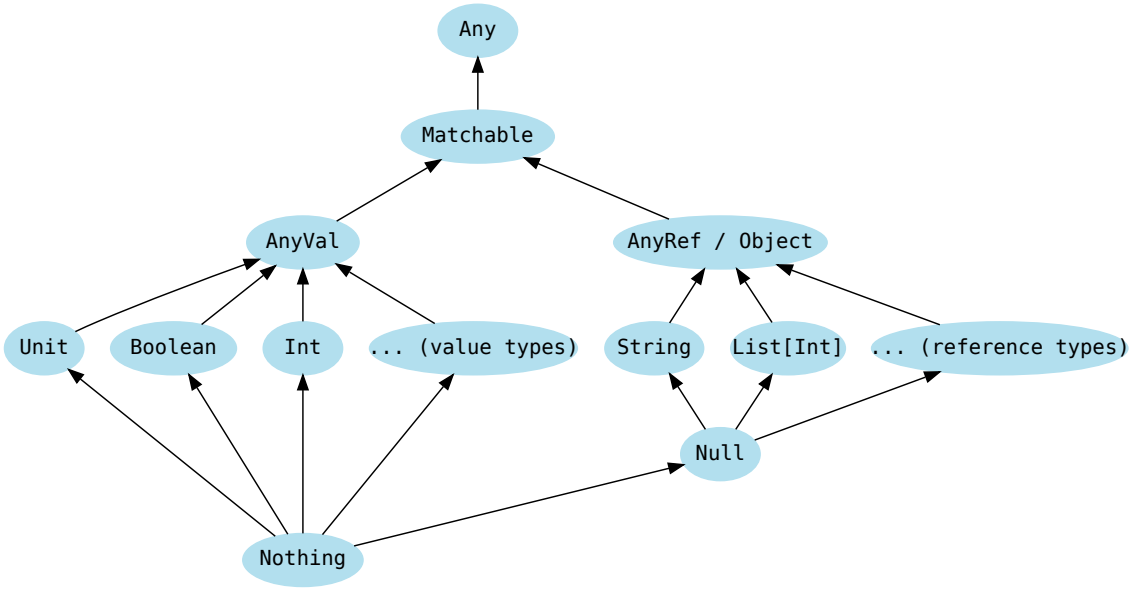


Figure 2.1: Scala 3 type hierarchy.

is `Matchable`, which marks values suitable for pattern matching. `Matchable` has two subtypes: `AnyVal`, a supertype for all value types, and `AnyRef`, a supertype for all reference types. `Null` is a subtype of all reference types, except when *explicit nulls* -feature is enabled and `Null` becomes a subtype of `Any`. Scala also has a bottom type `Nothing` that is a subtype of every type. No values of type `Nothing` can ever exist at runtime so the type reflects the absence of a value, for example in the case of an infinite recursion or loop, or when an expression throws an exception. The diagram in Figure 2.1 depicts Scala's type hierarchy.

Variance defines the rules on how the subtype relationships between parameterized types are dependent on the subtype relationship on the type on which they are parameterized. Variance has three variants: *invariance*, *covariance*, and *contravariance*. Invariance means that subtyping relationships present in type parameters are not applied to the parameterized type at all. Covariance states that the subtype relationship of the parameterized type is in the same direction as a type parameter's subtype relationship. Contravariance reverses the subtype relationships between pa-

rameterized types and their type parameters. When `Sub` is a subtype of `Super` and `F[_]` is any parameterized type, then

- Under covariance, `F[Sub]` is a subtype of `F[Super]`;
- Under contravariance, `F[Super]` is a subtype of `F[Sub]`; and
- Under invariance, `F[Sub]` and `F[Super]` have no subtyping relationship.

Covariance is applicable in parameterized types that contain, store, or produce values; the type parameter is in covariant position. Contravariance is applicable in the opposite situation, where values of the type parameter are consumed, i.e., the type parameter appears in a function parameter list, and is said to be in contravariant position. Invariance is useful in situations where it does not make sense for the parameterized type to have inheritance based on a type parameter, or when the parameterized type is used as the type of a mutable value, or the type parameter appears both in covariant and contravariant positions. A parametric type with multiple type parameters could declare each type parameter with different variance. For example, functions in Scala are contravariant in their input type(s) and covariant in their result type.

An infamous example of a mutable covariant type is the primitive array type in Java and C#. These arrays must perform a runtime type check when adding elements to the array, and throw an exception if the type of the element is not compatible with the array, as demonstrated in Listing 2. To avoid these mandatory casts and checks, mutable collections in Scala are invariant. Immutable collections and containers, such as `Option` or `Either` are covariant in Scala.

Programming languages differ in the way variance annotations are defined and used. Variance annotations in C# and Scala occur with the parameterized type. On the other hand, in Java one defines variance only when using a parameterized

Listing 2 Covariance in mutable types, like Java primitive array, is problematic.

```
String[] a = new String[1];
Object[] b = a; // String is a subtype of Object, so this is legal
b[0] = 1; // Runtime exception since cannot add Integer to String[]
```

type. The former is called *declaration-site* variance, demonstrated in Listing 3, and the latter is called *use-site* variance, demonstrated in Listing 4. Approaches deviating from these exist. For example, TypeScript tries to infer variance, but it has optional declaration-site annotations from version 4.7 onwards. Kotlin has declaration-site variance by default but it emulates some parts of use-site variance with type projections.

Listing 3 Scala uses declaration-site variance, where the variance of a parameterized type is denoted in its type definition.

```
class Invariant[A]      // Invariance is the default
class Covariant[+A]     // Covariance denoted with +
class Contravariant[-A] // Contravariance denoted with -
```

Listing 4 Java has use-site variance, where the desired variance is declared when using the parameterized type.

```
interface Supertype {}
interface Subtype extends Supertype {}

void invariant(List<Supertype> list) {
    /* Get and set list values */
}
void covariant(List<? extends Supertype> list) {
    /* Only get list values */
}
void contravariant(List<? super Subtype> list) {
    /* Only set list values */
}
```

Invariance is the default in Scala and it does not require an explicit annotation. Covariance is declared with a + sign before each type parameter. Since contravari-

ance could be seen as the opposite of covariance, it is denoted with a - sign.

Many features and principles from functional programming are not only available, but also encouraged in Scala. Pattern matching, first-class functions (Listing 5), and tail recursion are all supported and heavily utilized in idiomatic Scala programs. Immutable variables, collections and data-structures are the default way of writing Scala, even though mutable counterparts are also available. Functional data modeling is achieved with the use algebraic data types built into the language. Even though Scala embraces functional programming and imperative code is generally discouraged, introducing arbitrary side effects is possible.

Listing 5 Long and short form of anonymous functions in Scala.

```
val ns = List(1, 2, 3)

val mapped1 = ns.map(n => n + 1)
val mapped2 = ns.map(_ + 1) // Same as above in shorter form

val sum1 = ns.foldLeft(0)((x, y) => x + y)
val sum2 = ns.foldLeft(0)(_ + _) // Same as above in shorter form
```

In addition to ordinary functions, Scala has a specific function type called `PartialFunction` for representing functions that are not defined for all values of their input types. It is a subtype of the (normal) function type, to which it adds the method `isDefinedAt`, which determines if the function is defined for a given value. Listing 6 shows how to define and use partial functions.

Listing 6 Partial functions in Scala.

```
val someEvensMultipliedByTen: PartialFunction[Option[Int], Int] = {
  case Some(n) if n % 2 == 0 => n * 10
}

val opts  = List(None, Some(2), None, Some(3), Some(4))
val somes = opts.collect(someEvensMultiplied) // List(20, 40)
```

Some functional languages, such as Haskell, have a special syntax for monadic computations. Scala also provides this syntactic sugar in a form of `for`-comprehensions, demonstrated in Listing 21. The `for`-comprehension is compatible with any data type that has `map` and `flatMap` methods defined, such as `Option`, `Either`, and `ZIO` (Chapter 4). These required methods can be added to any type by using extension methods.

Extension methods, which allow adding methods to a class separately from its definition, are one of Scala's more advanced features. Other state-of-the-art features of Scala include operator overloading and infix operator- and method syntax, higher kinded and dependent types, type lambdas, as well as powerful meta programming capabilities. Scala 3 introduced more cutting edge features, such as automatic type class derivation and union and intersections types.

Probably the most distinguishing feature in Scala is its system of implicits and other contextual abstractions arising from that. A function can mark some of its parameters as implicit and the compiler will try to figure out these parameters from the enclosing scope by their type, without the programmer explicitly passing an argument for these parameters. Originally implicit parameters were introduced to achieve similar behavior as Haskell's type classes. Type classes are described in more detail in Appendix A. Implicits can, however, also be used for other purposes, such as implicit conversions, context propagation, extension methods, and proving subtyping relationships between generic type parameters at compile-time. [18]

Syntactically to use implicits, a function can mark some of its parameters as implicit with the keyword `using`. When the function is called, the compiler tries to find a value marked as implicit, with the keyword `given`, from the enclosing scope. If all requested values are found, they are automatically applied as arguments. If any of the implicit parameters is not found, compilation error is reported. Listing 7 shows the function `summon` that searches for an implicit value by type, demonstrating

the definition and use of implicit parameters.

Listing 7 Implicit resolution in Scala 3.

```
// Defined in the standard library
def summon[A](using a: A): A = a

// Defines an implicit value of type Int
given Int = 3

val three = summon[Int] // Finds the value 3 defined above

def implicitSum(a: Int)(using b: Int): Int =  a + b
val five = implicitSum(2) /*(compiler automatically inserts 3)*/
```

Another advanced feature utilizing implicit resolution is the ability of the Scala compiler to prove type equality or subtype relationships. The class `=:=[From, To]` is for type equality and `<:<[From, To]` for subtype relationship. Both classes extend a function `From => To`, and can be used to transform types. Types with two type parameters can be used as infix in Scala, for example type equality can be written `A =:= B`. When requesting an implicit parameter of either of the types above, Scala compiler synthesizes an instance if the type relationship holds, otherwise it reports a compilation error. The act of proving type relationships is said to be *witnessing*, and a common practice is to name the implicit parameter as *evidence*. The feature is useful, for example, when defining functions that make sense only for specific types, as demonstrated in Listing 8, where only nested `Maybe` types could be flattened.

Scala 2 and 3 are also differentiated by the introduction of intersection and union types in Scala 3. Intersection types are denoted with the `&` symbol and union types with `|`. Intersection `A & B` means that the resulting type has properties of both `A` **and** `B`. Union is the dual of intersection, and the resulting type of `A | B` is either `A` **or** `B`.

Intersection types are commutative, idempotent, and have `Any` as the identity

Listing 8 Scala compiler can prove (witness) a subtype relationship by providing implicit evidence.

```
enum Maybe[+A]:
  case Just(a: A)
  case Nothing

  def flatten[B](using evidence: A <:< Maybe[B]): Maybe[B] =
    this match
      case Just(a) => evidence(a)
      case Nothing => Nothing

Maybe.Just(Maybe.Just(1)).flatten // Compiles
Maybe.Just(1).flatten // Error: Cannot prove that Int <:< Maybe[B]
```

element. Commutativity means that the order of types included in the intersection does not matter — Scala considers permutations equal. Idempotency means that type intersectioned with itself is equal to the type itself. `Any` as the identity element means that the intersection of any type `A` with `Any` is equal to `A`, since all types themselves are subtypes of `Any`. Expressed as code, laws of intersection types can be proved with the Scala compiler:

- Commutativity: `summon[(A & B) =:= (B & A)]`
- Idempotency: `summon[A =:= (A & A)]`
- Identity: `summon[A =:= (A & Any)]`

Like intersection types, also union types are commutative, idempotent, and obey the identity laws. The identity element is `Nothing`: the union of any type `A` with `Nothing` is equal to `A`, since there are no values of type `Nothing`. Again expressed as code, the laws of union types proved by the Scala compiler are:

- Commutativity: `summon[(A | B) =:= (B | A)]` then
- Idempotency: `summon[A =:= (A | A)]`
- Identity: `summon[A =:= (A | Nothing)]`

3 Approaches for managing effects

This chapter presents different approaches of managing effects and how these approaches manifest as programming language features. These approaches include effect systems and unrestricted side effects, as well as more advanced techniques such as monadic effects, algebraic effects and handlers, and capability based effects.

3.1 Effect systems

The purpose of an effect system is to allow mixing effectful and pure code safely. The idea of an effect system is very similar to that of a type system. In some programming languages, a type system can be used to implement an effect system, such as in Java or C#, but in others they are two separate systems, such as in Unison [19] or Koka [20].

A type system sets the rules according to which functions, parameters, expressions, and, in some cases, objects can be composed. A static type system checks that these rules are obeyed before the program is run. An effect system enforces rules regarding the effects that expressions and statements have, and how these effects can interact with each other. Similarly to type systems, these interactions are checked statically at compile-time.

A type system infers or requires the programmer to specify the type of the values related to an expression. Analogously, an effect system infers or requires the programmer to specify the possible effects for every function/expression. Contrary

to type systems where an expression usually has just one type, an expression can produce zero or more different effects. Considering the possible effects related to an expression as a set, an empty set of effects denotes an expression that is free of effects.

Active research related to statically checking effects began in the mid 80s and 90s. Even earlier efforts in this direction were the Pascal extensions Euclid (in the 70s) and Ada (in the 80s) that separated side effecting procedures from pure functions [21]. The term effect system was introduced by Gifford and Lucassen [22] in 1986. Their idea was to assign different *effect classes* to different parts of a program. Gifford and Lucassen's paper proposed rules for how these different classes are allowed to interact with each other. For example, a pure function is not allowed to call a function that is labeled with a more permissive effect class. This allows the safe mixing of functional and imperative code while preserving equational reasoning of the functional parts and tracking possible effects of the imperative parts. In the system, the only effectful operations were related to allocating, mutating and reading memory locations. The goal was to determine what parts of the program could be run in parallel without changing the semantics of the program.

Probably the most widely known example of effect systems is checked exceptions in Java (Listing 9). This part of Java's type, or effect, system is concerned of tracking exceptions, more specifically where they are thrown and caught. If a method might throw an exception, the exception type must be declared in a `throws` clause in the method's type signature. The compiler forces any code that calls the method to either handle all declared exceptions or to add a `throws` clause to indicate that exceptions will bubble up. Checked exceptions have been widely criticized for making programming clumsy, and nowadays it is common for the whole feature to be circumvented when possible.

Since their introduction, effect systems have evolved significantly and gained

Listing 9 Checked exceptions in Java.

```
public byte[] readFile(String fileName) throws IOException {
    var file = new File(fileName);
    var is = new FileInputStream(file); // can throw IOException
    return is.readAllBytes();           // can throw IOException
}

public void catchIt() {
    try { var bytes = readFile("file.txt"); }
    catch (IOException exc) { /* Handle error  */ }
}

// Caller must handle IOException
public void declareIt() throws IOException {
    var bytes = readFile("file.txt");
}
```

more sophisticated features, such as the ability to track non-memory related effects like IO and exceptions. Several effect systems [19], [20], [23], [24] allow the user to define custom effect types. The research regarding effect systems is active, and several novel approaches and features are emerging.

One feature under active research and development is effect polymorphism [25]. The goal of effect polymorphism is to allow defining, in a safe way, functions that are polymorphic in the effect of their argument. This makes it possible to define, e.g., an effect polymorphic `map` function that accepts as an argument a transformation function and applies the transformation to elements in a context, such as a list. The challenge is to be able to define just a single a `map` function, per context, in a way that the input function can be either pure or have arbitrary effects that determine the effect of evaluating the `map` function.

Several researchers agree that discovering a practical solution to expressing effect polymorphism is crucial for the practical use of effect systems.

Odersky, Boruch-Gruszecki, Lee, *et al.* [25]:

> The problem is not lack of expressiveness — [effect] systems have been

> proposed and implemented for many quite exotic kinds of effects. Rather, the problem is simple lack of usability and flexibility, with particular difficulties in describing polymorphism.

Leijen [26]:

> In practice though we wish to simplify the types more and leave out 'obvious' polymorphism.

Lindley, McBride, and McLaughlin [6]:

> In designing Frank we have sought to maintain the benefits of effect polymorphism whilst avoiding the need to write effect variables in source code.

Languages with built-in effect systems [19], [20], [23], [24] usually include algebraic effects and handlers, which are discussed in more detail in Section 3.4. Library-level support for effect systems is commonly based on monadic effects, which are discussed in Section 3.3. Another approach for managing effects is *capability-based* effects, which are described in more detail in Section 3.5.

3.2 Unrestricted side effects

The most straightforward way to incorporate effects into a programming language is by not giving them any special treatment. This way pure expressions and effectful statements are treated equally and can be combined with each other in any way. The evaluation of any method, function or procedure can cause side effects to occur.

The origins of unrestricted side effects go back all the way to 50s and 60s when the first programming languages were created. Even in today's software industry, unrestricted side effects are the default way to incorporate effects into a programming

language. Virtually all mainstream programming languages allow unrestricted side effects in one way or another.

Not restricting side effects in any way gives the programmer a lot of freedom when implementing a program. By allowing effects in any expression, the language does not place constraints on how subprograms can be composed. This way is well-aligned with the imperative paradigm. However, the programmer is solely responsible for managing all the effects and making sure that they are compatible with each other. The language does not provide help in ensuring correct use of effects. This way of handling side effects is also not particularly expressive or modular. Creating reusable functions for common side-effecting operations, such as repeating an effect, defining a timeout or retrying, is hard or at least clumsy.

Listing 10 shows a higher-order function `map` for the `List` datatype, and demonstrates how it can be used with pure and effectful mapping functions. A similar function is used in the upcoming sections as a running example to demonstrate how such effectful mapping function can be implemented with other approaches for managing effects.

Listing 10 `List.map` function in Scala, where `f` can have arbitrary side effects.

```
extension [A](as: List[A])
  def map[B](f: A => B): List[B] =
    as match
      case head :: tail => f(head) :: tail.map(f)
      case Nil          => Nil

// The type signatures of all expressions are identical
// Even though the last one may throw an exception
val nums: List[Int] = List(1, 2, 3, 4, 5)
val pure: List[Int] = nums.map(n => n * 2)
val effectful: List[Int] = nums.map { n =>
  if n > 5 then throw RuntimeException("Too large") else n * 2
}
```

3.3 Monads

Of particular interest in this thesis is the algebraic structure monad. Algebraic structures are a concept that define functions that operate on some parametric type, or types, and are governed by algebraic laws. Algebraic structures are often studied through the lense of category theory, a branch of theoretical mathematics that studies objects, transformations between objects and relationships between different categories. In this thesis algebraic structures and monads in particular are approached from the perspective of computer science; the focus is on how monads are capable of encoding effects.

A functor is a transformation between two categories. In functional programming most, if not all, functors are endofunctors, which are transformations from one category back to the same category. In practice endofunctors wrap some other category and allow transforming the inner category while preserving the outer category. The list datatype is an example of an endofunctor, because it allows for applying transformations to elements in the list, resulting in a new list. A monad is a special kind of endofunctor that is capable of collapsing a nested endofunctor structure. In the case of lists, consider applying a transformation that would result in a nested list. A monad is capable of applying such a transformation in a way that the resulting list is not nested. Listings 11 and 12 show the definition of Functor type class in Haskell and Scala. Type classes are introduced in Appendix A.

Listing 11 Functor type class in Haskell.

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

The applicability of monads to programming was not discovered until the late 80s by Moggi [27], who showed how monads can define semantics of effectful pro-

Listing 12 Functor type class in Scala 3.

```
trait Functor[F[_]]:
  extension [A](fa: F[A]) def map[B](f: A => B): F[B]
```

grams. Moggi's proposed semantics extend lambda calculus in a pure way to support calculations previously considered to be impure. Later the idea of using monads to describe effectual computations was refined by Wadler [28], and Moggi [29] and Wadler [30]. The approach to effects based on monads can be summarized as describing computations as ordinary values.

Any data type can form a monad if it has at least two capabilities: lifting any value to the context of the monad (i.e., the data type), and sequentially composing computations that act on these values. Every computation in these sequences has access to the values that the preceding computations may have produced. These computations produce values that are inside a data type and succeeding computations have access to. Lifting and sequencing must adhere to monad laws in order for the data type to be considered a monad. Monad laws are discussed in more detail in Section 3.3.6.

In practice several data types naturally form a monad, such as `Array` in JavaScript with `of` function providing lifting and `flatMap` function providing sequencing [31]. Monads and other algebraic structures are often implemented as type classes, and writing programs consists of using operations provided by those type classes. This allows for writing abstract programs that work for any monad instance. The definitions of the Monad type class in Haskell and Scala are provided in Listings 13 and 14.

Listing 13 Monad type class in Haskell.

```
class Functor m => Monad m where
  return :: a -> m a
  ( >>= ) :: m a -> (a -> m b) -> m b
```

Listing 14 Monad type class in Scala 3.

```
trait Monad[F[_]] extends Functor[F]:
  def pure[A](a: A): F[A]
  extension [A](fa: F[A]) def flatMap[B](f: A => F[B]): F[B]
```

Composing programs of sequential instructions is nothing new compared to imperative programming. Monads, however, can control what effects are possible within computations and in their compositions. The data type (i.e., monad) provides the context in which the computations are performed, and thus defines the semantics of lifting and sequencing. Different monads have different semantics, which allows encoding different effects with monads. The usefulness of monads comes from the fact that sequencing computations one after another is such a fundamental operation in any effectful program. Monads abstract this operation, and allow for defining the meaning of sequentiality in the context of a specific monad. For example in a list monad, the semantics of sequencing is to perform the computation for every element in the list, and composing multiple lists will result in a cartesian product, as demonstrated in Listing 15. Examples of other monads and their semantics are given later in this chapter.

Listing 15 Monadic bind in list monad results in a cartesian product.

```
suits = ["Club", "Heart", "Diamond", "Spade"]
ranks = [2..14]

type Card = (String, Int)

-- [("Club", 2), ("Club", 3), ... ,("Spade", 13), ("Spade", 14)]
deck :: [Card]
deck =
  suits >>= \suit ->
  ranks >>= \rank ->
  return (suit, rank)
```

The naming of monad's functions is dependent on the programming language,

library and framework. The lifting function is usually called `pure`, `return`, `unit`, or `succeed`, and the sequencing function is called `bind`, `flatMap`, `chain`, or the symbolic alias `>>=`.

In addition to these mandatory functions, monads commonly define more specific functions that only make sense in the context of a particular monad. These functions make it easier and more convenient to use the capabilities of the monad, or possibly to change the behavior of computations in some way. Examples of such functions are presented below along with the introduction of specific monad types.

Monads are traditionally associated with statically typed languages, although nothing prevents their use in a dynamically typed language. In statically typed languages monads naturally work as an effect system by making explicit in the type system if and what effects are involved. When mixing multiple effects, type signatures can get quite chaotic. We will get back into this subject when discussing monad transformers.

3.3.1 Id

A trivial example of a monad is the identity, or `Id` monad. It simply encodes the effect of having no effect at all. Lifting values to monadic context is trivial since no lifting is required. The semantics of sequencing does not differ from conventional function application, as demonstrated in Listing 16.

Listing 16 Identity monad in Scala.

```
type Id[A] = A

given Monad[Id] with
  def pure[A](a: A): Id[A] = a
  extension [A](a: Id[A])
    def flatMap[B](f: A => Id[B]): Id[B] = f(a)
```

3.3.2 Either

`Either` monad encodes the effect of raising and handling exceptions when performing computations that might fail. Since `Either` is a monad, it enables the sequential composition of multiple possibly failing computations. Like the name suggests, computations in `Either` monads can either succeed with a value or fail with an exception. Either has similar short-circuiting semantics as throwing exceptions has in, e.g., Java. When the first exception is encountered, computations that follow will not be performed and the exception remains as the result of the computation. Usually `Either` provides combinators that can transform a failed computation into a successful one. This is semantically similar to catching exceptions. Unlike throwing and catching exceptions, `Either` makes it obvious in the type signature of the function that the computation the function describes has a possibility of failure.

In practice the `Either` data type is commonly implemented as a sum type of two variations: `Left` (exception) and `Right` (success). Usually implementations are right-biased, which, among other things, determines the semantics of monadic operations. To lift a value into `Either` monad, it is simply wrapped in `Right`. The meaning of sequencing in the case of `Right` is to pass the successful value to subsequent computations, whereas in the case of `Left` it is to return the failed exception as is and perform no computations. An example of `Either` monad implementation in Scala is given in Listing 17.

In order for `Either` to better support exception handling, several convenience functions are commonly defined for it. These functions are more specific than the monad structure admits, since they operate in a domain where the computation might produce different values. Next a few of the these functions are introduced in more detail.

One typical scenario in error handling is to define a fallback computation to be

Listing 17 Either monad in Scala.

```
enum Either[+E, +A]:
  case Left(e: E)
  case Right(a: A)

given [E]: Monad[[A] =>> Either[E, A]] with
  def pure[A](a: A): Either[E, A] = Right(a)
  extension [A](either: Either[E, A])
    def flatMap[B](f: A => Either[E, B]): Either[E, B] =
      either match
        case Left(e)  => Left(e)
        case Right(a) => f(a)
```

performed if the actual computation is unsuccessful. In Haskell this is achieved by utilizing an associative binary operation in `Semigroup` type class, which is defined as `(<>) :: Either e a -> Either e a -> Either e a`. In Scala similar semantics are made possible by `orElse` -method on an `Either` data type itself, with type signature: `def orElse[E1, A1](or: => Either[E1, A1]): Either[E1, A | A1]`. Because Scala 3 has union and subtypes, it is possible for the fallback computation to have different exception and success types as the original `Either`, here `A` is the success type of `Either`.

Another common operation in error handling is to transform the error type. There are some differences in the implementation of this functionality depending on the language. Haskell has `BiFunctor` type class where the function `first` allows applying transformations to the left side of `Either`. Scala has `LeftProjection`, which allows to perform monadic operations on the (left) error "channel" of the `Either`. `Either` in Scala also has `def swap: Either[A, E]` method that transforms a `Right` to `Left` and vice versa.

Possibly the most common operation in error handling is to derive some final value from a computation. Since the computation can have either failed or succeeded, both possibilities must be covered. This could be achieved by providing a function for both cases that transforms the corresponding value (failure or success)

to the same result type. In Haskell the type signature of this function is `either :: (a -> c) -> (b -> c) -> Either a b -> c` and in Scala it's `def fold[B](onLeft: E => B, onRight: A => B): B`.

3.3.3 Reader

The reader monad encodes the effect of describing a sequence of computations that require some shared context or environment in order to be evaluated. The idea closely resembles composing functions together by passing arguments from the parent to child functions. Instead of explicitly passing every parameter, the reader monad automatically threads the environment through computations. It is noteworthy that the reader monad itself is nothing more than a data structure that describes a computation. In order to retrieve the described result the computation must be executed by providing the environment it requires. Common use cases for reader monad are dependency injection and context sharing in deeply nested structures, such as function calls or component hierarchies in UI frameworks.

Listing 18 Reader monad in Scala.

```
case class Reader[-R, +A](run: R => A)

object Reader:
  def ask[R]: Reader[R, R] = Reader(r => r)

given [R]: Monad[[A] =>> Reader[R, A]] with
  def pure[A](a: A): Reader[R, A] = Reader(_ => a)
  extension [A](self: Reader[R, A])
    def flatMap[B](f: A => Reader[R, B]): Reader[R, B] =
      Reader(r => f(self.run(r)).run(r))
```

The implementation of the reader monad (Listing 18) is confusingly simple due to the fact that it is essentially just a wrapper for a function. It could be implemented as a single parameter function that receives the environment as an argument

and returns the result of the computation. Lifting a value to a reader monad is as simple as defining a function that ignores its argument and returns a specified value. The meaning of sequencing two reader computations together is to run both computations providing them with the same parameter [32].

Reader has a couple of common operations specific to it. One primitive operation is to retrieve the environment from the reader. The implementation is just the identity function, and the operation is often named `ask`, `get`, or `environment`. Another primitive operation is to actually run the computation the reader monad describes to get the final result from it. Running a reader monad is nothing more than providing the required environment, in some cases there is a helper function `run` or `runReader` to do just that.

3.3.4 IO

IO monad encodes the effect of performing side effects and possibly returning a value that depends on the side effect. This enables the implementation of programs that use, e.g., a console, file system, network or graphical user interface. It is common to also allow expressing mutability via IO monad. Also, IO monads usually provide a way to introduce and manage asynchrony, concurrency, and parallelism. With asynchronous operations comes the desire to define interruptions and timeouts, and handle asynchronous exceptions in a sound way, as discussed in Section 2.1.2.

Theoretical background of IO Monad is described by Peyton Jones and Wadler [5]. This work was published a couple of years after Moggi's initial discovery of using monads to model effects. IO monad was originally designed for Haskell, which is a lazily evaluated purely functional programming language. Due to being a lazy language, there is no explicit control flow — terms are evaluated only when absolutely required. Programming with side effects, however, requires that they are executed in a precisely defined order. Wadler and Peyton Jones describes the relationship

between lazy evaluation and side effects as follows: “laziness and side effects are fundamentally inimical”. Every expression in Haskell must be referentially transparent and programming with side effects is no exception. Modeling side effects with monads retains referential transparency and determines the execution order of expressions.

Wadler and Peyton Jones describe a parametric data type `IO a` that represents a possibly side effecting program that, **when executed**, returns a value of type `a`. In other words, `IO a` is an ordinary value that can be transformed by passing it into functions that return modified IO values. Also, a program may choose not to execute certain IO values even though they are defined. This idea of modeling side effecting programs as values turned out to be highly useful. It provides superior composability compared to programs with unrestricted side effects. For example it is possible to define combinators that work with every IO program and thus define behaviors like retrying, timeouts, error handling, parallelism and racing in a reusable manner.

IO monads and the idea of programs as values has been adopted to other languages than Haskell, including many impure and eagerly evaluated ones. Examples of such implementations are *ZIO* [33], *Cats Effect* [34] and *Monix* [35] in Scala, *Effect TS* [36] in JavaScript/TypeScript, *Arrow FX* [37] in Kotlin, *Missionary* [38] in Clojure, and *Eff* [39] and *Aff* [40] in PureScript.

Lifting a value into the IO monad means that no side effects are performed and the value is simply wrapped to IO. This bridges the cap between pure and impure worlds by making it possible to bring pure values into a context where describing side effects is possible. The meaning of sequencing is to create a description of two side effects that, when executed, are performed one after another. Like with all monads, the latter IO has access to the value produced by the preceding IO computation. A simple example implementation of IO monad is given in Listing 19.

Listing 19 Naive IO monad in Scala.

```
case class IO[A](run: () => A)

given Monad[IO] with
  def pure[A](a: A): IO[A] = IO(() => a)
  extension [A](io: IO[A])
    def flatMap[B](f: A => IO[B]): IO[B] =
      IO(() => f(io.run()).run())
```

The IO monad is fundamentally different from previously introduced monads, which can be implemented in a referentially transparent way. Since the IO monad encodes side effects it is inherently not referentially transparent, because the side effects must be executed *at some point*. To make it possible to write side-effecting programs in a purely functional way, the IO monad separates the *description* of side effects from the *execution* of side effects. Constructing a description of a side-effecting program is referentially transparent, while its execution is not; the latter is delayed, usually happening outside of "user-land" code.

To actually perform the side effects IO describes, there must be a way to interpret IO values into side effects they describe. This is usually the responsibility of the particular *runtime system*. In a purely functional programming language, the runtime cannot be implemented in the language itself. Impure languages have more flexibility in the way of implementing the runtime system, as well as how to encode the IO monad in the first place. Flexibility is useful: modern runtime systems with industry adoption are enormously complex and sophisticated, so that they can utilize the hardware as efficiently as possible to achieve the best performance possible.

Performance is really important, since the use of IO monad in a program is intrusive: any expression that references another expression that is evaluated in IO, must also be evaluated in IO. This is to be expected as there is no way to "peel off" the IO wrapper from an expression in a referentially transparent way, since that would mean executing the side effect. In programs written with the IO monad, the

runtime system can be seen as the central authority, as described in Section 2.1.

3.3.5 Syntax

The "usual" kind of code where functions are applied to values is called *direct style*. Programming with wrapped types (endofunctors), like monads, enforces a different style of syntax called *monadic style*. To perform operations on values in a monadic context, like combining multiple values together, one must use higher-order combinators, such as `map` and `flatMap`. The sequencing combinator will bind the value inside the monad to a variable that could be used in a function. Listing 20 compares the direct style with the monadic style, by the means of usual integer addition and integer addition in the Option monad.

Listing 20 Direct vs. monadic syntax in Scala.

```
// Direct style
val num1: Int = 3
val num2: Int = 4
val sum: Int  =
    num1 + num2
```

```
// Monadic style
val optionNum1: Option[Int] = Option(3)
val optionNum2: Option[Int] = Option(4)
val optionSum: Option[Int]  =
  optionNum1.flatMap(n1 =>
    optionNum2.map(n2 =>
      n1 + n2))
```

Programming with monads leads to numerous sequencing functions one after another. This can get verbose, and the intent of the code might be harder to see because it is obfuscated by the "monadic machinery". Some languages have built-in support for representing monadic computations in a more convenient way. Usually this comes in the form of special syntax for sequencing multiple monadic computations together with minimum boilerplate. The syntax is nothing more than syntactic sugar that the compiler converts to calls to monadic sequencing functions. Examples of such syntax are *Haskell do-notation* [41], *Scala for comprehensions* [42], *F# computation expressions* [43], and *OCaml Binding operators* [44]. Listing 21 compares Scala's for-comprehension syntax that desugars to sequence of `flatMaps`

and one final `map` function.

Listing 21 For-comprehension in Scala.

```
val optionSum: Option[Int] =
  optionNum1.flatMap(n1 =>
    optionNum2.map(n2 =>
      n1 + n2))
```

```
val optionSumFor: Option[Int] = for
    n1 <- optionNum1
    n2 <- optionNum2
  yield n1 + n2
```

A technique for programming in direct style with monadic effects while preserving the semantics of the specific monad has been proposed [45]. The technique is called *monadic reflection*, and it utilizes the fact that programs written in monadic style could be translated into programs written in Continuation Passing Style (CPS). The proposed technique requires from the programming language or platform a language-level support for first-class continuations/suspensions/coroutines. Monadic reflection requires for each monad an implementation of a type class with two operations: `reify` and `reflect` that wrap and unwrap values to and from the monadic context. The original motivation for monadic reflection was to support monadic effects in Scheme, but in practice monadic reflection has hardly gained any traction in any functional library or language. There has been, however, some recent research on how monadic reflection could work with capability-based effect tracking in Scala, and also a proof-of-concept implementation in Scala 3 [46], [47].

3.3.6 Monad laws

For a data type to form a monad, it must adhere to three laws, also known as the monad laws: associativity, left identity, and right identity. These laws are simply rules that the operations on a data type must follow. The laws precisely define the semantics of a data type and ensure that desired semantics are preserved when refactoring. In addition to their signatures, laws are what separate one algebraic structure from another. To be precise, an algebraic structure is totally defined by its

operations and the laws that govern these operations. Thus the definition of monad is an algebraic structure with two operations `pure` and `bind`, obeying the laws of associativity, left identity, and right identity, nothing more, nothing less. [48]

Listing 22 Monad associativity law in Scala.

```
def pure[A](a: A): Option[A] = Monad[Option].pure(a)

val num1: Option[Int] = Some(1)
val num2: Option[Int] = Some(2)
val num3: Option[Int] = Some(3)

val mustBeTrue = sumAll1 == sumAll2
```

```
def sumAll1: Option[Int] =
  num1.flatMap(n1 =>
    num2.flatMap(n2 =>
      num3.flatMap(n3 =>
        pure(n1 + n2 + n3))
    )
  )
```

```
def sumAll2: Option[Int] =
  num1.flatMap(n1 =>
    num2.flatMap(n2 =>
      pure(n1 + n2))
  ).flatMap(sum12 =>
    num3.flatMap(n3 =>
      pure(sum12 + n3))
  )
```

Associativity means that if there is a binary operation that is applied to three or more values, the order of application does not change the resulting value. In other words, the order of parentheses does not matter. Common examples of associative operations include integer addition and multiplication, string concatenation, and boolean `&&` and `||` operations. In the context of monads, associativity states that the semantics of sequencing are not dependent on the nesting of the `bind` operations. An example of this is provided in Listing 22.

Left and right identity laws define how lifting and sequencing must interoperate. Left identity states that if a value is lifted to a monadic context and then a function is applied to it using the sequencing operator, the result must be equal to just applying the function to the value without lifting it into the monadic context. Right identity states that if a value is lifted into a monadic context and sequenced into the

Listing 23 Monad identity laws in Scala.

```
def pure[A](a: A): Option[A] = Monad[Option].pure(a)
def f(n: Int)                = pure(n + 1)

// Left identity
val x: Int = 1
pure(x).flatMap(n => f(n)) == f(x)

// Right identity
val num: Option[Int] = pure(1)
num.flatMap(n => pure(n)) == num
```

lifting function, it must be equal to the original lifted value. An example of both identity laws is provided in Listing 23.

3.3.7 Monad transformers

So far we have gone through how monads can be used for encoding several side effects. However, in practice it is common that multiple effects need to be used in tandem. Practically all applications use the IO monad and they may desire to model exceptions and early termination with the Either monad, and access configuration or other context provided by the Reader monad. There is nothing to prevent manually stacking multiple monads to achieve all these functionalities.

Stacking multiple monads will lead to nested type signatures. The order of stacking is important, as the same types nested in different orders may imply totally different meanings. For example, `IO[Either[Error, Success]]` is a side effecting program that produces either a result of type `Success` or fails with an exception of type `Error`. On the other hand, an expression of type `Either[Error, IO[Success]]` is a program that will in the success case perform some side effects to produce a value of type `Success`, or fail with an exception of type `Error` without side effects.

Programming with nested monads leads to added boilerplate. To lift a value in to a nested monad, it must be manually wrapped with every monad in the correct

order. The programmer must manually thread the value inside monad layers through the program while preserving the nesting order and semantics of each monad. Every monad has slightly different semantics, so implementation details differ depending on the monad type. Listing 24 demonstrates the required syntax when programming with nested IO and Either monads.

Listing 24 Syntactic overhead of nesting Either and IO monads.

```
def fn(str: String): IO[Either[Unit, Int]] = ???

val ioEitherString: IO[Either[Unit, String]] = IO(Right("foo"))

val ioEitherInt: IO[Either[Unit, Int]] =
  ioEitherString.flatMap(either =>
    either.fold(
      error => IO(Left(error)),
      success => fn(success),
    )
  )
```

In addition to obfuscating the intent, manually implementing all of this functionality is a burden to the programmer and a possible source of bugs. Sometimes the cause of bugs could be highly subtle, for example when using Either for error handling inside IO, as shown in Listing 25. The programmer might be relying on the short-circuiting semantics of Either but when it is used inside the IO monad, the error is silently swallowed. It is even possible that the return type of `mayFail` was initially `IO[Unit]`, and it was later refactored to also include an error case. In this situation, the compiler does not report an error since discarding values is allowed. As there are arbitrarily many ways to nest monads, the number of similar possible bugs is also large.

Nesting monads also comes with performance considerations. Calls to monadic functions must propagate through every layer of nesting, which increases indirection

Listing 25 Subtle bugs not causing early termination or compilation error.

```
def mayFail: IO[Either[String, Unit]]  = ???
def wontFail: IO[Either[Nothing, Int]] = ???

val program: IO[Either[String, Int]] =
  for
                    // Type of _ is Either[String, Unit]
    _   <- mayFail  // Even if this line evaluates to Left
    res <- wontFail // ... this line will still be executed
  yield res
```

and the number of function calls. Also memory consumption increases because each nested monad necessarily consumes some amount of memory. The exact magnitude of performance implications depends on the language, platform, and runtime environment.

Listing 26 EitherT monad transformer in Scala.

```
case class EitherT[F[_], E, A](effect: F[Either[E, A]])

given [E, F[_]: Monad]: Monad[[A] =>> EitherT[F, E, A]] with
  def pure[A](a: A): EitherT[F, E, A] =
    EitherT(Monad[F].pure(Right(a)))

  extension [A](self: EitherT[F, E, A])
    def flatMap[B](f: A => EitherT[F, E, B]): EitherT[F, E, B] =
      EitherT(
        self.effect.flatMap {
          case Left(e)  => Monad[F].pure(Left(e))
          case Right(a) => f(a).effect
        },
      )
```

Monad transformers can avoid nested monads and help compose multiple monads into one. A Monad transformer is simply a wrapper for one monad that gives it also the semantics of another monad, just like nested monads. Like every monad, the composed monad must obey the monad laws. There is no universal way to compose monads, each monad must have its own monad transformer instance. For some

monad pairs composition is meaningless, or it is not possible to define a monad transformer.

The nested monad in Listing 24, `IO[Either[E, A]]`, is isomorphic to `EitherT[IO, E, A]` (defined in Listing 26), which is a monad transformer for Either monad applied to IO. This monad is capable of encoding side effects as well as terminating early in the presence of errors. Listing 27 shows an identical program to that in Listing 25 but one that does not suffer from the issues described earlier. This is since `EitherT` composes with any other monad with short-circuiting semantics.

Listing 27 Usage of EitherT monad transformer with IO monad.

```
def mayFail: EitherT[IO, String, Unit]  = ???
def wontFail: EitherT[IO, Nothing, Int] = ???

val program: EitherT[IO, String, Int] =
  for
                       // Type of _ is Unit
    _     <- mayFail  // If this line evaluates to Left
    value <- wontFail // This line won't be executed
  yield value
```

Monad transformers alleviate some of the issues encountered when nesting monads manually. There is less syntactic overhead since the monad transformer threads the values through the monad stack and does all the required wrapping and unwrapping. However, many of the problems with nested monads are also present in monad transformers. The order of nesting is still significant, performance considerations are similar and every monad requires a unique implementation.

Because Scala has subtyping, it emposes some unique constraints to monad transformers. `EitherT` defined in Listing 26 is invariant on the monad it composes. With this definition the code in Listing 27 will not compile since `mayFail` and `wontFail` do not have identical type signatures. To overcome this issue, there are multiple solutions, each with their pros and cons. One might define `EitherT` to require the

composed monad to be covariant. This has the obvious downside that it restricts what monads are compatible with `EitherT`. An other option would be to define widening operators on invariant `EitherT`, but that would place a burden on the programmer who would have to explicitly invoke those methods. Both options are demonstrated in Listing 28.

Listing 28 EitherT leftWiden method.

```
case class CovariantEitherT[F[+_], +E, +A](effect: F[Either[E, A]])

case class EitherT[F[_], E, A](effect: F[Either[E, A]]):
  def leftWiden[E1 >: E]: EitherT[F, E1, A] =
    this.asInstanceOf[EitherT[F, E1, A]]
```

3.3.8 Polymorphism

Many higher-order combinators found in collections and other data types, such as `map`, `filter`, `zip`, and `fold`, do not work when the input function is monadic. This means that a specific monadic counterpart is required for each of combinator. The implementations of these combinators differ considerably from the implementations of the corresponding pure operators. However, the implementations can usually be generalized to work with every monad, including monad transformers. A convention originating from Haskell is to suffix such combinators with `M` to indicate that it is the monadic version of the combinator. Listing 29 defines the effect polymorphic monadic `mapM` operator for `List` that is compatible with any monad.

Similarly, looping and branching constructs require their own monadic versions, such as `ifM` and `whileM`, when the predicate is evaluated in a monad. The need for separate monadic combinators is definitely one of the weaknesses of monads, and a possible stumbling block for a newcomer.

Monads provide a referentially transparent way of modeling effects. As a result,

Listing 29 Monadic `mapM` function for `List` in Scala.

```
extension [A](as: List[A])
  def mapM[F[_]: Monad, B](f: A => F[B]): F[List[B]] =
    as match
      case Nil => Monad[F].pure(List.empty)
      case head :: tail =>
        for
          b  <- f(head)
          bs <- tail.mapM(f)
        yield b :: bs

val nums: List[Int] = List(1, 2, 3, 4, 5)
val effectful: Either[String, List[Int]] =
  nums.mapM { n =>
    if n > 5 then Left("Too large") else Right(n * 2)
  }
```

programs written using monads are modular and can be safely refactored. Expressivity is also high; there are many operators, and new ones can be easily implemented in terms of existing ones. However, monads largely determine how programs should be written. They force monadic syntax instead of direct one. Many existing combinators and control structures require a monadic counterpart. Also, composing different effects together is not straightforward since it requires nesting monads or using monad transformers.

3.4 Algebraic effects and handlers

Algebraic effects and handlers are one of the most recent approaches and fields of research on the subject of purely functional effectful programming. Algebraic effects take the approach that there is a variety of different types of effects and every effect type has a finite set of *operations* that define potentially impure abilities. To interpret each operation, one must provide a *handler* for every effectful operation. Operations define the interface of the effect, while handlers define the semantics of

each effect and operation.

The notion of "algebraic operations" was introduced by Plotkin and Power [49] in 2001 and they refined the idea in [50] and [51]. The idea of handlers accompanying algebraic effects was first presented by Plotkin and Pretnar [52] in 2009 and later Plotkin and Pretnar [53] in 2013. The idea was similar to what Moggi discovered in [29], but Plotkin and Power considered operations to be primitive instead being derived from the monadic context.

The availability of algebraic effects and handlers is mostly, at least currently, in strict/eagerly evaluated purely functional programming languages. The idea of transferring the control to an effect handler does not fit the model of lazily evaluated languages naturally, since languages with lazy evaluation do not have explicit control flow. [54]

3.4.1 Existing languages and libraries

Algebraic effects can be implemented as a library or a language-level feature. There are several libraries that provide some support for algebraic effects in languages that do not have native support for them, like Idris Effects [55], Haskell Extensible effects [56] and F# AlgEff [57]. In the 2010s the theory of algebraic effects evolved in to several research languages such as Eff [58], Koka [20], Frank [23], Links [59], and Effekt [60]. The appearance of algebraic effects in non-research languages has only happened in the past few years, with Unison [19] and OCaml [24].

Unison is a programming language with several out-of-the-ordinary features, including *abilities*, which are an implementation of algebraic effects from Frank [23]. Unison has had alpha and beta versions since 2019 and it is currently aiming to achieve commercial adoption. OCaml version 5.0 [61] (released in December 2022) includes language-level support for algebraic effects and handlers. As can be seen, currently algebraic effects are a new concept with little to no experience from in-

dustry.

3.4.2 Theory of handlers

When a program encounters an effect operation, its execution is halted, and the control is transferred to the closest handler provided for that specific operation. The handler may also receive some parameters from the program in the process of taking over the execution. After the transfer of control, it is solely the responsibility of the handler to decide how the program will continue.

The idea of effects being interaction between sub-programs and a central authority, described in Section 2.1, fits algebraic effects naturally. The parts of the program that call effect operations of algebraic effects are the sub-programs and the handlers are the central authority. Compared to monads, algebraic effects take a different approach. Pure values are separate from effectful computations, which are defined as effect operations and performed by handlers. The concept is powerful enough to implement all previously mentioned monads and even many of the more complicated control structures, built-in to many languages, like try-catch, iterators, and async/await. [54]

A common way of handling an effect is to transform it to another effect or data type. Many times higher-level effects are implemented in terms of lower level effects, and finally the most primitive effects, such as IO, are provided by runtime. Eventually this forms a graph of effects and handlers that depend on each other [62]. Providing an expression with an effect handler it requires is said to "discharge" the effect from the expression. In order to safely evaluate an expression all of its effects must be discharged.

Handlers have a way to continue executing the program, and optionally apply a transformation function to the final value of the expression they handle. However, it is totally up to the specific handler to decide how and if to continue the execution or

whether to apply the final transformation. This way the handler has the full power to decide how to act. It may continue the execution and, depending on the operation, supply a value to continue with, or it may decide to terminate the execution and continue by executing a different part of the program instead. The handler may even decide to execute a continuation multiple times and possibly collect all results of the continuations to a list.

It is worth noting that the handlers required by a well-formed program can be changed without having to change the program code in any way. This could have interesting implications in for example multi-platform development, where one could abstract platform-specific operations to effects and provide different handlers depending on the platform. For example one could provide an effect interface for concurrency, which would have drastically different handler implementations in a single-threaded environment, such as JavaScript, compared to multi-threaded environment, like JVM. This would be opaque from the perspective of the programmer who is using the effect interface.

3.4.3 Handlers in practice

Languages and libraries that implement algebraic effects provide effect handlers access to a continuation function that, when called, resumes the execution of the program from where it was transferred to the handler in the first place. In other words, the continuation is a function that represents the remaining of the program after the effect is handled.

There are several ways to implement handlers in a language. Handler can be either *deep* or *shallow*. A deep handler handles all effects of specific type in an expression, while a shallow handler only handles the first effect of its corresponding type. A shallow handler can usually be converted to a deep handler by applying it recursively. The continuation provided to the handler can be either *single-shot* or

multi-shot. A single-shot continuation can be invoked only a single time, whereas a multi-shot continuation can be invoked many times. A handler usually handles only a single effect, but if the language supports *multihandlers*, the same handler can handle several effects at once.

Handlers in Unison are shallow with multi-shot continuations. A handler can continue executing the program by calling the continuation function available when pattern matching against the possible effect constructors. The syntax for defining a handler for a single effect operation is as follows:

`{ <operation> <param1, ... , paramN> -> <continuation> } -> <result>`.

Matching a final transformation, or the pure case, is defined with a simple pattern:

`{ <operation-result> } -> <handler-result>`.

Listing 30 Exception ability and handler in Unison.

```
structural ability Exception e where
  raise : e -> a

toOptional : '{Exception e} a -> Optional a
toOptional mightThrow =
  handle !mightThrow with cases
    { raise e -> c }  -> None
    { a }             -> Some a
```

Continuations in Koka's handlers are multi-shot, like in Unison, but the handlers are deep, unlike in Unison. In Koka an operation handler is defined with the syntax: `<operation>(<param1, ... , paramN>) -> <result>`, and the continuation is implicitly in scope via the keyword `resume`. The final transformation is defined with `return(<operation-result>) -> <handler-result>`

Listings 30 and 31 demonstrate how to define an effect and a handler, as well as how to use the final transformation function when implementing effect handlers. They define an effect type `Exception` that is capable of interrupting a program by

Listing 31 Exception effect and handler in Koka.

```
effect exception
  ctl raise (exc : e) : a

fun to-maybe(might-throw : () -> <exception|x> a) // : x maybe<a>
  with handler
    raise(e)  -> Nothing
    return(a) -> Just(a)
  might-throw()
```

raising an exception of type `e`, while the uninterrupted program would have resulted in a value of type `a`. The handlers discharge the effect by translating it to the data type `Optional a`/`Maybe a` by converting a `raise` operation to `None`/`Nothing` and utilizing the final transformation to convert a value of type `a` to `Some a`/`Just a`.

Listing 32 Definition and usage of Choice effect in Unison.

```
structural ability Choice where
  choose : Boolean

pickNumber : '{Choice} Nat
pickNumber = do
  if choose then
    if choose then 12 else 21
  else
    if choose then 34 else 43
```

Listing 32 defines an effect `Choice` that has a single operation `choose` that results in a `Boolean`. The function `pickNumber` selects a number based on the results of the `choose` operation. The code that uses the effect does not enforce how the choosing operation should be implemented, but it works with any implementation.

A possible handler implementation for the `Choice` effect could be a handler that always chooses the same `Boolean` value. Listing 33 gives an example of such a handler with two helper handlers, `alwaysTrue` and `alwaysFalse` that always choose the corresponding value.

Listing 33 Effect handlers for Choice that always result in constant value.

```
constantChoice : Boolean -> '{Choice} a -> {} a
constantChoice choice thunk =
  handle !thunk with cases -- shallow handler is applied recursively
    { choose -> resume }  -> constantChoice choice '(resume choice)
    { a }                 -> a

alwaysTrue  : '{Choice} a -> {} a
alwaysTrue = constantChoice true

alwaysFalse : '{Choice} a -> {} a
alwaysFalse = constantChoice false

alwaysTrue pickNumber  -- 12
alwaysFalse pickNumber -- 43
```

Another possible handler implementation is one that collects all possible results in a list. The handler resumes the program multiple times, two times for every `choose` operation to be precise. An example of such an implementation is given in Listing 34.

Listing 34 Effect handler for Choice that collects all possible results.

```
collectAll : '{Choice} a -> {} [a]
collectAll thunk =
  collectHandler : Request Choice a -> [a]
  collectHandler = cases
    { choose -> resume }  ->
      (handle resume true with collectHandler) ++
      (handle resume false with collectHandler)
    { a } -> [a]

  handle !thunk with collectHandler

collectAll pickNumber -- [12, 21, 34, 43]
```

3.4.4 Effect typing

Programming with algebraic effects clearly separates effectful computations from values, which makes a language with algebraic effects a good candidate for separate type **and** effect systems, which were discussed in Section 3.1. All effectful expressions must be provided with corresponding handlers before execution, and by utilizing an effect system, this check can be made statically. Algebraic effects themselves do not require a static type system, but practically all current programming languages with first-class algebraic effects are equipped with an effect system.

When an expression references an effectful operation, the effect system adds that effect to the set of effects associated with the expression. On the other hand, when an effect handler is provided for an expression, the effect system can remove the effect from the set of effects for that specific expression, and possibly add new effects if the implementation of the handler references other effects. Usually algebraic effects can be inferred and do not need to be mentioned in the source code.

Previous examples demonstrate how effect system and algebraic effects cooperate. In Listing 32, `pickNumber` is an expression that evaluates to a natural number and references the `Choice` ability/effect. The referenced effect is reflected in the type signature of the expression. In Listings 33 and 34 handler functions for the `Choice` effect are defined. Constant handlers `alwaysTrue` and `alwaysFalse` simply discharge the effect from expressions. The discharging of the effect is evident in the type signature, as it changes from `{Choice} a` to `{} a`, which indicates that the expression does not reference any unhandled effects. The collecting handler `collectAll` discharges the effect, and also changes the type of the expression.

Unlike monads, algebraic effects naturally compose with one another. An expression can reference any number of effects and those effects are simply added to the set of effects associated with the expression. Similarly to nested monads and monad transformers, the order in which handlers are applied is significant and changing

the order of handlers might significantly alter the semantics of the program. Listing 35 shows the effect signature when an expression references multiple effects, in this case `Choice` and `Exception` effects.

Listing 35 Effect composition in Unison.

```
failingNumber : '{Choice, Exception Text} Nat
failingNumber = do
  if choose then 34
  else raise "Better luck next time"
```

Algebraic effects with handlers usually make it possible to achieve effect polymorphism by defining an *effect variable* that represents a generic effect type, or lack thereof. Listings 36 (Unison) and 37 (Koka) both demonstrate this by giving an implementation of the `map` function for lists, as well as introducing its usage. When `nums` are mapped with a function without any effects, the resulting list `pure` is free of effects. On the other hand, when `nums` are mapped with an effectful function, the resulting list `effectful` depends on the `Exception` effect.

Listing 36 Effect polymorphic `map` function in Unison.

```
-- {e} is the polymorphic effect variable
map : (a -> {e} b) -> [a] -> {e} [b]
map f = cases
    head +: tail  -> f head +: map f tail
    []            -> []

nums  : [Nat]
nums = [1, 2, 3, 4, 5]

pure : [Nat]
pure =  map (n -> n * 2) nums

effectful : '{Exception Text} [Nat]
effectful _ = map (n -> if n > 5 then raise "Nope" else n * 2) nums
```

Listing 37 Effect polymorphic `map` function in Koka.

```
// e is the polymorphic effect variable
fun map(lst : list<a>, f : a -> e b) : e list<b>
  match lst
    Cons(head, tail)  -> Cons(f(head), map(tail, f))
    Nil               -> Nil

val nums: list<int> = [1, 2, 3, 4, 5]

val pure: list<int> = map(nums, fn(n) n * 2)

fun effectful(): exception list<int>
  map(nums, fn(n) if n > 5 then raise("Too large") else n * 2)
```

Algebraic effects provide high expressive power and good modularity with effectful computations. Programs can be written in direct style, albeit handlers must be implemented in a special way. Algebraic effects also offer seamless composability between different effects and different effects can be combined freely. Effects can be locally introduced and eliminated. The usually included effect system helps with refactoring by ensuring that if an existing expression is modified to have a new effect, it is handled appropriately.

Unlike monads, algebraic effects do not offer equational reasoning and it is not always safe to replace an expression with its value. Algebraic effects are still an active field of research with many open questions regarding, for example, shallow vs. deep handlers, single vs. multi-shot continuations, and multihandlers.

3.5 Capability based effects

Recently another approach, called *capabilities*, for describing effects has emerged. Compared to algebraic effects and handlers, capabilities view effects differently; instead of denoting an expression with effects, they place the requirement that the evaluation context must contain a *capability* for an effect. The distinction between

the two approaches may seem subtle, but capabilities seem to provide better developer ergonomics by reducing the need to specify effect types in code. [63]

One can think algebraic effects as rising *outwards* from effectful expressions, while capability based effects (more specifically, capabilities) as going *inwards* from the surrounding context towards effectful expressions. This way of modeling effects with capabilities seems promising. It should be possible to define effect polymorphic higher-order functions without the need to mention any effect variables in their type signature. For example, an effectful `List.map` function in Scala would have the signature `def map(f : A => B): List [B]`, which is exactly the same as that of the current `map` function in Scala. This is possible because `map` does not require any *additional* capabilities in addition to capabilities required by `f`. If `f` requires any capabilities, the context in which `map` is called must provide those capabilities to `f`. [25]

The idea of using capabilities to model effects is recent and research on this idea is in its infancy. Programming languages supporting capability based effects are experimental and rare. Effekt [60] is a research language with capability based effects with handlers. There is also an ongoing research project to study how effects could be managed with capabilities in Scala 3.

3.5.1 Capture checking

Scala 3 is based on a research language called Dotty. The name Dotty comes from Dependent object types (DOT), which are the theoretical foundation behind Scala 3 [64]. During writing this thesis, an experimental and work-in-progress feature called Capture checking [65] was added to Dotty and later to Scala 3 nightly builds. Capture checking is a language feature based on capabilities. The idea of capture checking is to enable effectful programming in direct style (discussed in Section 3.3.5), yet tracking effects in the type system and providing strong static guarantees

of the correctness of the program.

The initial version of capture checking is based on the work of Odersky, Boruch-Gruszecki, Lee, *et al.* [25] on modeling polymorphic effects with capabilities. Odersky, Boruch-Gruszecki, Lee, *et al.* criticize the currently widely used ways of managing effects, such as Java's checked exceptions and monads, arguing that they lack in both usability and flexibility, and result in complex and duplicated code. They conclude that this is due to the transitive nature of effects in function call chains, combined with the classical type-systematic approach that "characterize the shape of values but not their free variables", and suggest that modeling effects with capabilities may circumvent the problems they described.

The goal of capture checking is to address many of the limitations in effectful programming. These include how to solve the "What color is your function" problem [66], how to express effect polymorphism, how to combine manual and automatic memory management, how to express high-level concurrency and parallelism safely, and how to migrate already existing programs to use capture checking [67]. Active research on capture checking focuses on the usability aspect of static effect tracking, which likely will evolve around effect polymorphism, inferring the captured capabilities, direct style of programming, and in general minimizing the overall syntactic overhead.

Capture checking extends the idea of capabilities by tracking what capabilities are closed over, i.e. captured, by an expression. This means that a capability can be just a normal value, like any other variable in a program. In the context of capture checking, an expression is pure if it does not capture any capabilities. To make programming with capabilities easier, capabilities can be implicitly passed to expressions, instead of requiring one to explicitly thread capabilities through a program. The implicit system in Scala should be well suited for this task.

Capture checking aims to address effects such as throwing exceptions, IO, mu-

tability, suspending computations and continuations. Capture checking should also be applicable to resource handling. Resources, such as file handles, network connections, or memory must be acquired before use, and disposed afterward to free up allocated resources. A resource has a *lifetime* in which it can be used, and the use of an already disposed resource should be prevented. Resource lifetimes and rules should preferably be enforced statically by the type system. There are close connections between capture checking and linear type systems, of which Rust lifetimes is a well-known example [68].

Odersky's research group is actively working on capabilities, as evidenced by a recent large grant [69]. This research project is called Caprese (Capabilities for resources and effects), and its goal is a universal theory of resources and effects based on capabilities. It will be interesting to see the results from Odersky's group in the coming years.

4 ZIO

This chapter focuses on ZIO and its features, which are discussed in the context of the theory covered in the previous chapters. First a brief history and overview of ZIO is presented. The rest of the chapter is organized into several sections to analyze ZIO's features. The first section introduces the basic operators used in ZIO. The second section covers error handling by presenting ZIO's error model and its error handling operators. The next section explores ZIO's most distinguishing feature: the environment as well as operators and data types related to it. The chapter then moves onto resource management, detailing how ZIO facilitates safe resource handling. The fifth section is about concurrency in ZIO, showcasing the concurrency model and related operators. The chapter concludes with a summary of ZIO and highlights how ZIO is able to address issues presented in the earlier chapters.

ZIO [33] is an open-source Scala library/framework for managing side effects and modeling asynchronous and concurrent programs in a purely functional way. The development of ZIO started in 2017 by John De Goes. The first stable release of the library took place in the summer of 2020, so ZIO is quite new. At the time of writing this, the most recent version is 2.0.10, released in March 2023. De Goes and Adam Fraser, a core contributor to the project, co-authored a book about ZIO called Zionomicon [70], which is extensively used as a reference in this chapter.

The ZIO ecosystem consists of dozens of official and several more third-party libraries that include, among other things, testing, streaming, logging, caching,

JSON-parsing, database interaction, and HTTP servers and clients. Despite being a new library, many large companies, including DHL, eBay and Zalando are using ZIO in production. There is also a very active and quickly expanding ecosystem around ZIO, which has libraries and interoperability packages with other libraries and ecosystems. Today, ZIO is one of the fastest growing ecosystems in Scala.

ZIO is based on monadic effects but it also takes influence from algebraic effects and handlers. ZIO aims to provide a pragmatic, purely functional, type safe, easily testable and declarative API for asynchronous and concurrent effectful programming. The idea of ZIO is to combine multiple effects into a single monad and thus avoid the need for monad transformers.

The library is built around `ZIO[-R, +E, +A]` monad, which has three type parameters. `E` and `A` parameters represent the error and success channels, much like in Either monad. The functionality of Either monad is just one aspect of ZIO: ZIO is also capable of describing asynchronous and side-effecting computations. The `R` parameter describes the requirements, environment, or context, needed to perform the computation captured by the monad. In this sense ZIO is similar to the reader monad, but again, the reader monad is only one aspect of ZIO, and also this reader aspect has some extra capabilities that are introduced later in this chapter. Drastically simplifying, a ZIO computation can be seen as function from an environment to either an error or a success value: `R => Either[E, A]`. The idea of ZIO's three type parameters is that it should be possible to encode most, if not all, of the practically useful effects in a single monad.

ZIO provides type aliases for common variants, among others:

- `type UIO[A] = ZIO[Any, Nothing, A]` has no requirements and cannot fail.
- `type IO[E, A] = ZIO[Any, E, A]` has no requirements and can fail with `E`.
- `type URIO[R, A] = ZIO[R, Nothing, A]` has requirement `R` and cannot fail.

Since ZIO is a monadic effect system, all computations are values that can be transformed with functions. This makes it easy to implement combinators for modifying ZIO-values, thus changing the behavior of the described computation. ZIO provides numerous built-in combinators for error handling, context management, dependency injection, concurrency, retrying and repeating, scheduling, memoizing, resource management, and more. It is also easy to implement complex custom combinators in terms of existing ones.

ZIO's approach to functional programming is pragmatic, aiming to be easy to learn, even for programmers without prior theoretical knowledge about functional programming concepts. Even though the library has strong theoretical foundations in functional programming, the goal is to not have them surface in the public API more than necessary. Using ZIO does not require knowledge of concepts like type classes or monad transformers, even though the former is utilized internally. Function naming mostly avoids terms originating from category theory, symbolic operators, and naming conventions from Haskell. For example, functions corresponding to Haskell's `sequence`, `traverse`, and `bracket`, are named `collectAll`, `foreach`, and `acquireReleaseWith` in ZIO to make them easier to understand. There is a naming convention originating from Haskell where monadic combinators, such as `foldM`, `ifM`, and `replicateM`, are suffixed with `M`. The meaning of `M` might not be obvious to newcomers and ZIO aims to make it clearer by naming these combinators as `foldZIO`, `ifZIO`, and `replicateZIO`. Haskell's convention to suffix the names of combinators that discard their result with `_`, e.g. `sequence_` or `traverse_`, is not followed: the respective ZIO names are `collecAllDiscard` and `foreachDiscard`.

ZIO also takes advantage of multiple advanced features of Scala to make the API more convinient to use. The implicit system is used to provide context information for tracing, derive type class instances and prove type relationships. Dependent types are used, for example, to destructure nested tuples when zipping together

multiple ZIO values. There are several combinators that only make sense with specific success or error types. These operators utilize implicit evidence provided by the Scala compiler to make sure they are used appropriately. Examples of such cases are error handling operators that are only applicable with effects that can actually fail. Metaprogramming is utilized, for example, in dependency injection: the dependency graph is resolved and constructed at compile time, failing compilation if any of the required dependencies is not provided.

Monadic programming in Scala has traditionally suffered from the lack of type inference due to subtyping, forcing the programmer to explicitly write type annotations. Prior to ZIO, many functional programming libraries in Scala implemented their monads with invariant type variables because of these issues related to subtyping, and because of issues related to type inference with monad transformers, mentioned in Section 3.3.7. Since ZIO does not use monad transformers, it does not suffer from limitations associated with them. ZIO embraces the subtyping and variance of Scala by declaring the error and success types covariant and the environment type as contravariant. This makes type inference a lot more effective: explicit type definitions are rarely required when combining ZIO effects with different type parameters.

4.1 Basic operators

One of ZIO's most used classes of operators are constructors that create ZIO values. Like every monad, ZIO also has a lifting function `ZIO.succeed`. In addition to lifting pure values, it also enables the lifting of non-fallible side effects to ZIO. For lifting side effects that might throw exceptions, `ZIO.attempt` is used. To create failed ZIO effects, functions `ZIO.fail` or `ZIO.die` are commonly used. ZIO constructors use lazy, by-name parameters to delay the execution of unintentional side effects until the ZIO effect is actually executed. Error handling in ZIO is discussed in more

detail in Section 4.2. Constructors for data types from Scala standard library like `Option` and `Either` exist as well. Usage of the most common ZIO constructors is demonstrated in Listing 38.

Listing 38 Common ZIO constructors.

```
val pureValue: UIO[Int]    = ZIO.succeed(1)
val sideEffect: UIO[Unit] = ZIO.succeed(println("Hello World!"))

val either: Either[String, Int] = ???
val fromEither: IO[String, Int] = ZIO.fromEither(either)

val effect: IO[Throwable, Array[Byte]] = ZIO.attempt {
  val file = new File("numbers.txt")
  val is   = new FileInputStream(file) // can throw IOException
  is.readAllBytes()                    // can throw IOException
}

val error: IO[String, Nothing] = ZIO.fail("Error")
val defect: UIO[Nothing]       = ZIO.die(new Exception("Error"))
```

The simplest operators on ZIO monads are the ones that include a single ZIO value. The operator for applying a pure transformation to a value inside ZIO is implemented by the `map` function. The operator to discard the value in ZIO and map it to a constant value is the function called `as`. A common debugging operator for peeking the value inside ZIO without changing the value is called `tap`. ZIO also has a specific `debug` operator that will print the value inside ZIO with the provided prefix. The use of the above mentioned operators are demonstrated in Listing 39.

Listing 39 Common ZIO transform operators.

```
val one: UIO[Int]        = ZIO.succeed(1)
val two: UIO[Int]        = one.map(_ + 1)
val discardOne: UIO[Int] = one.as(34) // same as map(_ => 34)

one.tap(n => ZIO.succeed(println(s"One: $n")))
one.debug("One") // Same as above, prints "One: "34"
```

An other much used category of operators are the ones combining two ZIO values together. The `flatMap` function present in all monads naturally exists in ZIO as well. For combining two independent ZIO workflows together, there is a whole family of *zipping* operators. Unlike monadic composition via `flatMap`, when zipping values together the second value cannot use the value produced by the first one. The most simple zipping operator, `zip`, simply runs both ZIOs from left to right and combines their results in a tuple. The `zipWith` allows for supplying a function to combine the left and right value into the resulting ZIO. Sometimes a ZIO is only evaluated because of the effect it produces, and its return value is not needed. For these puproses `zipRight` and `zipLeft` operators are useful. These combinators evaluate both ZIOs from left to right, but retain only the return value of the side indicated by the operator name. Right and left zipping combinators also have symbolic aliases, generally quite rare in ZIO, `*>` and `<*`, where the arrow points to the side whose value is returned. The combinators for two ZIOs are demonstrated in Listing 40.

Listing 40 Common binary combinators in ZIO.

```
val num  = ZIO.succeed(34)
val str  = ZIO.succeed("A string value")
val tell = ZIO.succeed(println("Hello World"))

// All three below are semantically equal
val v1: UIO[(Int, String)] = num.flatMap(n => str.map(s => (n, s)))
val v2: UIO[(Int, String)] = num.zipWith(str)((n, s) => (n, s))
val v3: UIO[(Int, String)] = num.zip(str)

val zipRight: UIO[Int] = tell.zipRight(num)
val zipLeft: UIO[Int]  = num.zipLeft(tell)

// Evaluation order: tell, num, tell. Returns the value of num
val toldTwoTimes: UIO[Int] = tell *> num <* tell
```

When there is a need to combine more than two ZIOs togheter, for example to combine a collection of ZIO effects together, there are operators for that as

well. An effectful for loop is provided by the `ZIO.foreach` function, which takes a collection of values and a function that performs some effectful computation for each value. The operator performs all computations and returns a collection of results. A similar operator is `collectAll`, which receives a collection of ZIO computations, and returns a collection containing the results of the computations. Both operators are demonstrated in Listing 41. In order to effectfully fold over a collection of values, ZIO provides, among others, `mergeAll`, `reduceAll`, `foldLeft`, and `foldRight` functions to compute a single summary value from a collection.

Listing 41 Common combinators for multiple values in ZIO.

```
def findById(id: Int): UIO[Result] = ???
def combineResults(total: Int, result: Result): Int = ???

val ids = List(1, 2, 3)

val found1: UIO[List[Result]] = ZIO.foreach(ids)(findById(_))
val found2: UIO[List[Result]] = ZIO.collectAll(ids.map(findById(_)))

val combined: UIO[Int] =
  ZIO.mergeAll(ids.map(findById(_)))(0)(combineResults(_, _))
```

4.2 Error handling

Proper error handling is essential in any non-trivial application, as discussed in Section 2.1.2. Failures in ZIO are described in a referentially transparent way by returning values that represent the error, instead of throwing exceptions. Like other monads capable of encoding exceptions, ZIO stops the execution of the success channel on the first encountered error, until the error is handled with one of the error handling combinators. Much of the errors and their handling are tracked in types, making it possible to have static proofs that all declared errors are handled. ZIO advocates its error model, which is promised not to lose any errors, even if

they are asynchronous, parallel, caused by interruptions, or exceptions thrown by finalizers.

ZIO divides failures into three categories: errors, defects and fatal errors. Fatal errors, such as `OutOfMemoryError`, are thrown by the runtime platform (usually JVM), and result in immediate termination of the application, and thus are not very interesting in this context. The two remaining error types describe failures that the programmer can interact with. Errors are represented as the `E` parameter in ZIO, and are tracked in types. `Nothing` has a cardinality of zero, which proves that ZIO with `Nothing` in the error channel cannot produce a failing ZIO, thus the computation is infallible. Defects are not reflected in types, and practically any ZIO can produce a defect when executed. The type of a defect is always Java's `Throwable`.

The error channel should be used for business errors that are expected to sometimes happen and can be handled in a meaningful way and recovered from. On the other hand, defects are failures that are unexpected, or errors for which there is no meaningful way to handle or recover from. Because Scala programs are mostly run on the JVM, where exceptions could be thrown anywhere, ZIO runtime catches all thrown exceptions and reports them as defects. This makes it easier to integrate ZIO code with code not written with ZIO, such as Java-libraries where throwing exceptions is the de-facto error reporting and handling strategy. Roughly speaking, logical exceptions (discussed in Section 2.1.2) are usually errors, while technical exceptions are usually defects.

Errors in ZIO are internally represented with a data type `Cause`, which is an instance of the algebraic structure *semiring*, that is capable of capturing the full chain of possible failures, including errors, defects, and interruptions, sequential or parallel. This data type also keeps track of a trace that lead to the failure described by a specific cause. Such a trace is similar to an ordinary stack trace

but it is able to describe operations across asynchronous boundaries and does not expose unnecessary implementation details of the underlying runtime. ZIO provides operators for interacting with the `Cause` data type directly, but usually higher level operators that work with error or defect types are preferred. The definition of a simplified `Cause` data type and an example of its use is provided in Listing 42.

Listing 42 Cause data type captures the full cause of failures.

```
// Cause in ZIO also includes traces omitted here
enum Cause[+E]:
  case Empty
  case Fail(value: E)
  case Die(value: Throwable)
  case Both(left: Cause[E], right: Cause[E])
  case Then(left: Cause[E], right: Cause[E])
  case Interrupt(fiberId: FiberId)

val a = ZIO.dieMessage("A")
val b = ZIO.fail("B").ensuring(ZIO.sleep(5.millis).timeout(1.milli))
a.zipPar(b).cause.debug
// Cause.Both(
//   Cause.Die(java.lang.RuntimeException("A")),
//   Cause.Then(
//     Cause.Fail("B"),
//     Cause.Interrupt(<FiberId of the interrupting fiber>),
//   ),
// )
```

When two ZIOs are composed together, the composed ZIO could fail either with the error from the first or the error from the second one. The order in which the error types appear should not matter and all permutations consisting of the same types should be equal, i.e., composition is commutative. If the two ZIOs share the same error type, the resulting ZIO has the equal error type with the original ZIOs, i.e., composition is idempotent. If either of the two ZIOs cannot fail (the error type is `Nothing`), its error type does not contribute to the resulting error type, i.e., composition has `Nothing` as the identity element. Union types in Scala 3 naturally

have all these properties and precisely express composition of error types: an error type represents a set of possible errors, composition is then set union and `Nothing` represents the empty set. If the execution of a ZIO fails, the error is **one of** the errors in the set of possible errors. Listing 43 demonstrates the accumulation of errors in types.

Listing 43 Typed error accumulation when composing multiple ZIO values.

```
val num1: ZIO[Any, ErrorA, Int]           = ???
val num2: ZIO[Any, ErrorA, Int]           = ???
val num3: ZIO[Any, ErrorB, Int]           = ???
val doSomething: ZIO[Any, Nothing, Unit] = ???

// 'ErrorA' is included only once in the error type
// 'Nothing' is not included at all in the error type
val composed: ZIO[Any, ErrorA | ErrorB, Int] =
  for
    n1 <- num1        // ErrorA
    n2 <- num2        // ErrorA
    _  <- doSomething // Nothing
    n3 <- num3        // ErrorB
  yield n1 + n2 + n3
```

Ideally there would be no need to explicitly add a type annotation about the error type when composing ZIOs together, and the programmer could simply rely on type inference. The Scala compiler tries automatically to *unify* the types, i.e., find the closest common supertype between the composed ZIO values. The `E` parameter in ZIO is covariant, which is essential for type inference when combining multiple ZIOs together. Because `Nothing` is a subtype of every type, ZIO that has `Nothing` in the `E` channel is automatically considered to be a subtype of ZIO that has the same `R` and `A` type parameters.

There are many similar operators for working with values in the error channel than there are for working in the success channel. For example `mapError`, `flatMapError` and `tapError` all work similarly to their success channel counter-

parts. Some of the most common error handling operators include catching some or all errors, providing a fallback computation, or folding over error and success values. The operators `catchAll` and `catchSome` behave like catch blocks in a try-catch clause, and as the names suggest, they handle, respectively, a subset or all errors. The `orElse` operator makes it possible to define a fallback computation whose success and error is used in the case when the original ZIO fails. ZIO has many variations of `fold` for pure and effectful folding that are semantically similar to folding an `Either`, discussed more in Section 3.3.2. These basic error handling operators are demonstrated Listing 44.

Listing 44 Basic error handling operators in ZIO.

```
type Error = ErrorA | ErrorB | ErrorC

val mayFail: IO[Error, Int] = ???

val handled: IO[Nothing, Int] = mayFail.catchAll(e => ZIO.succeed(0))

val someHandled: IO[Error, Int] =
  mayFail.catchSome { case _: ErrorA => ZIO.succeed(34) }

val folded: UIO[Int] = mayFail.fold(e => -1, n => n + 10)

val withFallback: IO[Nothing, Int] = mayFail.orElse(ZIO.succeed(0))
```

In addition to the `try-catch` like semantics described above, `try-finally` is a common pattern in imperative programming. Regardless whether the code in the `try` block throws exceptions or not, the code in `finally` block is guaranteed to be executed. The underlying idea is that there are one or more finalizers that need to be run after a certain block of code is executed. ZIO also supports this pattern with several operators that are guaranteed to execute the finalizers even in the presence of parallelism, asynchrony, concurrency, interruption, errors, and defects. Listing 45 demonstrates the basic finalizing operator `ensuring` that executes the

specified finalizer regardless of any kind of failure or interruption. Other, higher level, operators for `try-finally` like semantics are discussed more thoroughly in Section 4.4 about resource management.

Listing 45 Basic finalizer operator `ensuring` in ZIO.

```
val finalizer = ZIO.succeed(println("Finalizer executed"))

// The finalizer is executed once after each ZIO below is executed
val success: UIO[Int]    = ZIO.succeed(1).ensuring(finalizer)
val error: IO[Int, Int]  = ZIO.fail(42).ensuring(finalizer)
val defect: UIO[Nothing] = ZIO.dieMessage("No").ensuring(finalizer)

val interruption: UIO[Unit] = for
  fiber <- ZIO.sleep(1.second).ensuring(finalizer).fork
  _     <- fiber.interrupt // The finalizer is executed here
yield ()
```

The fact that ZIO has two-typed channels of output values (error and success), makes it possible to create interesting combinators that switch values between the two channels. An operator that simply swaps the channels with each other is `flip`. Another way to expose errors in the success channel is the `either` operator that converts a fallible ZIO to `ZIO[R, Nothing, Either[E, A]]`, resulting in an effect that cannot fail, but instead surfaces errors with `Either` in the success channel. The dual of `either` is the operator `absolve` that separates `Either` cases from the success channel to error and success channels of ZIO. The `Cause` data type can also be exposed in the success channel with the `cause` operator, making it possible to operate on errors, defects and interruptions at the same time. The reverse operator is `uncause`: it hides the `Cause` data type from the type signature. Type signatures of the above mentioned operators can be seen in Listing 46.

The same exception might be considered an error at some abstraction level and a defect at some other abstraction level. For example when implementing functionality

Listing 46 Operators for swapping values between error and success channels.

```
trait ZIO[-R, +E, +A]:
  def flip: ZIO[R, A, E]

  def either: ZIO[R, Nothing, Either[E, A]]
  def absolve[E1 >: E, B](using A <:< Either[E1, B]): ZIO[R, E1, B]

  def cause: ZIO[R, Nothing, Cause[E]]
  def uncause[E1 >: E](using A <:< Cause[E1]): ZIO[R, E1, Unit]
```

that is directly interacting with a relational database, it would be sensible to treat `SQLException` as an error and expose it in the `E` parameter. On the other hand, higher level abstractions that use this functionality, like repositories or services, usually should not declare `SQLException` in their signature, and treat it as a defect.

ZIO contains operators for switching values from the error channel to defect channel and the other way around. A simple way to convert errors to defects is to consider all errors as defects, which could be achieved with the `orDie` operator that switches all errors from the error channel to the defect channel. In order to have more control of what errors to retain, the `refineOrDie` operators are useful. They allow picking desired errors by providing a type parameter or a partial function, and the operator converts all errors not matching the type parameter or partial function to defects. To go the other way around and switch values from the defect channel to error channel, the `resurrect` operator moves all defects to errors and `unrefine` moves some defects to errors, like `refine` but the other way around. Listing 47 demonstrates the usage of these operators.

Sometimes when an error occurs, it can be resolved by retrying the operation that produced the error. Retries in ZIO only apply when the failure is in the error channel, and not in the defect channel. If one would like to retry even when a defect happens, the defect must first be surfaced to the error channel. Probably the simplest retry operator is `eventually`, which will retry forever until the operation

Listing 47 ZIO operators for switching between errors and failures.

```
val readFile: IO[Throwable, Array[Byte]] =
  ZIO.attempt(new FileInputStream("file.txt").readAllBytes())

val allErrorsToDefects: IO[Nothing, Array[Byte]] = readFile.orDie

val someErrorsToDefects: IO[FileNotFoundException, Array[Byte]] =
  readFile.refineToOrDie[FileNotFoundException]

val allDefectsToErrors: IO[Throwable, Array[Byte]] =
  allErrorsToDefects.resurrect

val someDefectsToFailure: IO[FileNotFoundException, Array[Byte]] =
  allErrorsToDefects.unrefineTo[FileNotFoundException]
```

succeeds. Usually it makes sense to limit the number of retries, and the `retryN` operator enables just that. For specifying custom rules when to retry and when to give up, ZIO has `retryUntil` and `retryWhile` operators that take a predicate as a parameter and retry according to that predicate. Basic retry operators are demonstrated in Listing 48.

Listing 48 Basic retry operators in ZIO.

```
val readFile: IO[Throwable, Array[Byte]] =
  ZIO.attempt(new FileInputStream("file.txt").readAllBytes())

val retryForever: UIO[Array[Byte]]             = readFile.eventually
val retryFiveTimes: IO[Throwable, Array[Byte]] = readFile.retryN(5)

val retryUnlessFileNotFound: IO[Throwable, Array[Byte]] =
  readFile.retryUntil {
    case _: FileNotFoundException => true
    case _                        => false
  }
```

Instead of immediately retrying, a common way is to schedule the retries with a delay in order to allow the error to resolve. ZIO has a specific data type `Schedule` for describing retry policies and other scheduling use cases. It is a purely functional

and composable data type capable of describing complicated schedules. In addition to retries, schedules are also applicable for describing the repetition and scheduling the execution of ZIO computations. Listing 49 introduces some basic `Schedule` constructors and combinators. When retrying ZIO with a delay, one might desire to limit the total time the computation can take, which is achieved with the `timeout` operator.

Listing 49 Schedule data type in ZIO.

```
Schedule.spaced(7.millis) // Constant delay between every computation
Schedule.fixed(7.millis)  // Computations start at constant intervals
Schedule.fibonacci(2.millis)   // 2ms | 4ms | 6ms | 10ms | 16ms
Schedule.exponential(2.millis) // 2ms | 4ms | 8ms | 16ms | 32ms

Schedule.forever   // Schedule always wants to continue
Schedule.stop      // Schedule that never wants to continue
Schedule.recurs(5) // Schedule that wants to continue 5 times

left ++ right // First left schedule to complection, then right
left && right // Recurs when both schedules want to continue
left || right // Recurs when either schedule wants to continue
```

4.3 Environment

Arguably the most distinguishing feature about ZIO is its environment, or the `R` type. The possibility to express environmental/contextual requirements of a computation plays a big part in the fact that ZIO can encode several effects in one monad, thus mostly eliminating the need for monad transformers. ZIO environment is similar to a reader monad, but there are a couple of key differences. Unlike the reader monad whose only effect is to provide read-only access to some context, the ZIO environment is just one of the effects that can be expressed with ZIO. Also, the environment type composes naturally when combining multiple ZIO values. The environment type in ZIO can be changed from one type to another, similar to indexed reader monads [71].

It is also possible to locally both introduce and eliminate (some or all) environmental requirements.

Recall that a mental model of a `ZIO[R, E, A]` is function `R => Either[E, A]`. Before a ZIO can be executed the required environment must be provided, just like a function must be provided with the arguments before it can be evaluated. A ZIO workflow that has no environmental requirements has `Any` as its environment type. Function `f: Any => Either[E, A]` function accepts *anything* as its argument. It can be called, for example, by providing the unit value `f(())`, a number `f(42)`, or a string `f("foo")` as an argument. The analogy applies to ZIO, where `ZIO[Any, E, A]` is ready to be executed without providing any environment.

Listing 50 Environment types accumulate when composing multiple ZIO values.

```
val num1: ZIO[String, Nothing, Int] = ???
val num2: ZIO[Int, Nothing, Int]    = ???
val num3: ZIO[Any, Nothing, Int]    = ???

// 'Any' does not appear in the environment type
val composed: ZIO[String & Int, Nothing, Int] =
  for
    n1 <- num1
    n2 <- num2
    n3 <- num3
  yield n1 + n2 + n3
```

When combining ZIO values together, the resulting ZIO naturally has environmental requirements from *all* the combined ZIOs. Similarly to error accumulation, composition should be commutative and have `Any` as its identity element. Scala 3 intersection types have these properties and they thus express environment composition accurately. Listing 50 demonstrates the accumulation of environment types when composing ZIO values.

Basic operations for interacting with the environment are adding requirements to it and eliminating all or part of the requirements. It is also possible to translate

one environmental requirement to another. A value from the environment can be accessed with `ZIO.service` function, which is similar to the `ask` function in Reader monad, with the exception that `ZIO.service` can return a part of the environment instead of the entire environment. Listing 51 demonstrates different operators for accessing the environment and adding environmental requirements.

Listing 51 Operators for adding requirements or accessing the ZIO environment.

```
// Same as 'ask' in reader monad
val ask: ZIO[String, Nothing, String] =
  ZIO.service[String]

// Eqivalent to: ZIO.service[String].map(_.length)
val askAndMap: ZIO[String, Nothing, Int] =
  ZIO.serviceWith[String](_.length)

// Equivalent to: ZIO.service[Random].flatMap(_.nextInt)
val askAndFlatMap: ZIO[Random, Nothing, Int] =
  ZIO.serviceWithZIO[Random](_.nextInt)
```

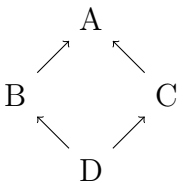
4.3.1 ZLayer

Environmental requirements in ZIO are provided in the form of a purely functional data type called `ZLayer`. `ZLayer[RIn, E, ROut]` has the same three type parameters as ZIO itself, and it is thus capable of expressing effectful, asynchronous, and possibly failing construction of requirements. The `RIn` parameter in `ZLayer` represents dependencies that are required in order to construct a value of type `ROut`. These dependencies between layers form a graph of dependencies, like the one demonstrated in Listing 52.

The environment for a ZIO workflow is provided with operators such as `provide` (provide all requirements), `provideSome` (provide a part of requirements), and `provideLayer` (convert existing requirements into other requirements), that take

Listing 52 Dependencies between `ZLayer`s form a graph.

```
val layerA: ZLayer[Any,    Nothing, A] = ???
val layerB: ZLayer[A,      Nothing, B] = ???
val layerC: ZLayer[A,      Nothing, C] = ???
val layerD: ZLayer[B & C, Nothing, D] = ???
```



`ZLayer`(s) as their argument. Also the `apply` method in `ZLayer` can be used to eliminate requirements from a ZIO workflow. ZIO can resolve the dependency graph with compiler macros, for example in `ZIO.provide` and `ZLayer.make` functions, and raise a compilation error if all required dependencies are not provided. Different ways of providing layers is demonstrated in Listing 53.

Listing 53 Providing layers from Listing 52 to a ZIO.

```
// ZLayer.make and ZIO.provide resolve the dependency graph
val useD: ZIO[D, Nothing, Int] = ZIO.service[D].as(34)

val layer: ZLayer[Any, Nothing, D] =
  ZLayer.make[D](layerA, layerB, layerC, layerD)

val provided1: ZIO[Any, Nothing, Int] = useD.provideLayer(layer)
val provided2: ZIO[Any, Nothing, Int] = layer(useD) // layer.apply
val provided3: ZIO[Any, Nothing, Int] =
  useD.provide(layerA, layerB, layerC, layerD)
```

Dependency injection is a design pattern that helps writing loosely coupled programs. The goal is to separate the logic of building a service from that of using the service. Such separation also makes it possible to provide different implementations of a service intended for different situations. `ZLayer`s along with ZIO environment are the basis of dependency injection in ZIO. Dependency injection in ZIO is resolved statically at compile time, so programs with missing dependencies will not compile.

4.3.2 ZEnvironment

The example in Listing 50 has a ZIO value `composed` that has `String & Int` as the environment type. No values of this type can exist at runtime, since there is no value that is both `String` and `Int`, so the type is only sensible at compile time. These kinds of types that only exist at compile time are sometimes called *phantom types* [72].

The `R` type parameter in ZIO is a phantom type, and therefore represents the required types only at compile time. However, every type present in the environment type intersection must have a corresponding value at runtime. This is achieved with a data type called `ZEnvironment[R]`, which can be seen as a map associating every type in the environment type intersection to a value, as demonstrated in Listing 54.

Listing 54 `ZEnvironment` contains the required environment for ZIO workflow.

```
// Can be thought of as: Map(Int -> 42, String -> "foo")
val environment: ZEnvironment[String & Int] =
  ZEnvironment.empty
    .add[Int](42)       // Explicit types here are not required
    .add[String]("foo") // but they are added for clarity

// Values from the environment can be accessed by their type
val int    = environment.get[Int]    // 42
val string = environment.get[String] // "foo"
```

With this knowledge, the mental model of ZIO can be updated to be `ZEnvironment[R] => Either[E, A]`. Since `ZEnvironment` is a low-level data type used internally to represent the environmental requirements of ZIO, its direct use is not advised, higher-level operators and data types such as `ZLayer` should be used instead.

4.3.3 Use cases

The ZIO environment can be used in many ways. In addition to providing read-only data to computations like reader monad does, it can describe mutable state, safe resource management (discussed more in Section 4.4), or dependency-injection. One could also use environmental requirement as a marker that a certain ZIO computation must be run in a specific context. Another common use case is to define combinators that translate a certain environmental requirement into another.

Probably the most basic use case of ZIO environment is to provide some static data/context, which the computation can use as it wishes. An example of such data is configuration data in a web application, possibly containing a URL for performing http requests. This is demonstrated in Listing 55.

Listing 55 Static data can be provided to computations with the ZIO environment.

```
case class Configuration(url: String)

val useConfiguration: ZIO[Configuration, Nothing, Result] =
  ZIO.serviceWithZIO[Configuration](conf => makeRequest(conf.url))

val configurationLayer: ZLayer[Any, Nothing, Configuration] =
  ZLayer.succeed(Configuration(url = "https://example.com"))

val configurationProvided: ZIO[Any, Nothing, Result] =
  configurationLayer(useConfiguration)
```

Another use case is to encode mutable state in the environment, similar to a monad transformer for `State` monad. This is achieved with a data type describing mutable references evaluated in the ZIO monad, such as `Ref` or `ZState`, that is purposefully built for this use case. State can be accessed with `ZIO.getState` function, which also adds a state requirement to the environment. State requirement can be eliminated using the `ZIO.stateful` operator by providing the initial state. Listing 56 demonstrates the usage of these operators in a stateful computation. A nice

byproduct of encoding state in the ZIO environment is that the environment can carry several different states at the same time, as long as the states are of different types.

Listing 56 Mutable state can be encoded with the environment in ZIO.

```
val statefulComputation: URIO[ZState[Int], Int] = for
  state <- ZIO.getState[Int]       // Access state
  _     <- ZIO.setState(state + 1) // Modify state
yield state

val statefulProgram: URIO[ZState[Int], Unit] = for
  _ <- statefulComputation.debug("First")
  _ <- statefulComputation.debug("Second")
  _ <- ZIO.getState[Int].debug("Last")
yield ()

// Provide initial state (0) to the stateful computation
// When executed prints: "First: 0", "Second: 1", "Last: 2"
val stateProvided: UIO[Unit] = ZIO.stateful(0)(statefulProgram)
```

Environmental requirements can be converted from one type to another by eliminating one requirement and adding a new one. Listing 57 demonstrates one such situation. In that example `businessLogic` requires a `UserSession` from the environment. There is a `UserService` that can validate a token (`String` in this case) and succeed with `UserSession`, or fail validation with `TokenError`. For example, in the context of a web application, a token could be extracted from a http request. The helper function `UserService.withSessionFromToken` takes two parameters: a token and a ZIO computation that requires `UserSession` from the environment, and returns a ZIO computation that requires `UserService` from the environment, which will be used to validate the token. If the validation is successful a `UserSession` is provided to the computation. If validating the token fails, the whole computation fails with `TokenError`, and the computation received as a parameter will not be executed. The possibility that validating the token might fail can be observed from the

fact that `TokenError` is added to the error type of the returned ZIO computation.

Listing 57 ZIO environment can be used to translate a contextual requirement to other requirement.

```
trait UserService:
  def validate(token: String): IO[TokenError, UserSession]

object UserService:
  def withSessionFromToken[R: Tag, E, A](token: String)(
      needsSession: ZIO[R & UserSession, E, A]
  ): ZIO[R & UserService, E | TokenError, A] =
    val session = ZIO.serviceWithZIO[UserService](_.validate(token))
    val layer   = ZLayer(session) // Create a ZLayer from ZIO value
    layer(needsSession) // Provide session as layer to ZIO workflow

val businessLogic: ZIO[UserSession, Nothing, Result] = ???

val program: ZIO[UserService, TokenError, Result] = for
  token  <- getToken // For example from a HTTP request
  result <- UserService.withSessionFromToken(token) { businessLogic }
yield result
```

The power of the environment type comes from the fact that it supports many different overlapping use cases. For example, configuration, state, and sessions can coexist in the environment without interfering with each other. The environment can be provided locally to a specific computation or globally to the entire program.

4.3.4 Similarity to algebraic effects

It may not be immediately obvious how ZIO is similar to algebraic effects and handlers. However, if we consider that each type in the environment is representing a specific effect, adding or interacting with environmental requirements represents an effectful operation, and removing an environmental requirement with `ZLayer` represents handling an effect, the similarity is imminent.

Like handlers in algebraic effects, `ZLayers` can handle (or discharge) the effect by removing it altogether, or it can translate one effect into another. Similarly

to handlers in algebraic effects, `ZLayer`s commonly form a graph of dependencies between other `ZLayer`s. ZIO environment composes in similar way as effects in a language that natively supports algebraic effects and handlers, such as Unison.

With `ZLayer`s it is possible to define a polymorphic handler, which only handles a subset of all effects in a specific expression. In practice this means that a `ZLayer` eliminates only a part of the environment, while leaving the rest in place. Listing 58 demonstrates the mentioned similarity and polymorphic handlers.

Listing 58 ZIO workflows and ZLayers can be seen as really similar to algebraic effects and handlers.

```
trait ZLayer[-RIn, +E, +ROut]:
  // Environmental requirement of type ROut is removed, and RIn is
  // added to the ZIO received as parameter.
  def apply[R, E1, A](
      zio: ZIO[ROut & R, E1, A]
  ): ZIO[RIn & R, E1 | E, A]

val handleAtoB: ZLayer[B, Nothing, A] = ??? // Changes A -> B
val handleB: ZLayer[Any, Nothing, B]  = ??? // Eliminates B

val effect: ZIO[A & Boolean, Nothing, Int] = ???

// ZLayer is polymorphic in the type of environmental requirement
// Here it removes B, adds A, and leaves Boolean as is
val handledA: ZIO[B & Boolean, Nothing, Int] = handleAtoB(effect)

// ZLayer (handler) for B does not have any requirements, so B is
// removed from the environment entirely, leaving only Boolean
val handledB: ZIO[Boolean, Nothing, Int] = handleB(handledA)
```

Effect polymorphism (demonstrated in Listings 29, 36 and 37), however, is limited since every ZIO computation is evaluated in a monadic context. Also handlers (i.e., `ZLayer`s) in ZIO are not as expressive, since they do not receive a continuation to the rest of the program, like algebraic effect handlers do.

4.4 Resource management

At a high level, resource management consists of three parts: acquiring resources, using resources and releasing resources after they are no longer needed. Numerous things can be viewed as resources that need to be acquired and released: concurrency or database locks, allocated memory, open file handles or network sockets, connections from a connection pool, or spawned processes/threads. Even a database transaction is a special kind of resource where releasing it either commits the transaction or rolls it back.

Important for the correct behavior of programs is that once a resource is acquired, it must be released, even if using the resource raises an exception or fails in some other way. This behavior can be described with a contextual data type that is added to the environment when resources are acquired, and that stays in the environment as long as there are resources that need to be released. A consequence of this is that acquired resources are visible in the type signatures that describe computations and the compiler is able to help in making sure that acquired resources are actually released, but not too soon.

Safe resource management in ZIO relies on information threaded through computations in the ZIO environment. In ZIO, the data type describing the lifetime of resources is called `Scope`. In principle, a `Scope` is very simple. It has only two operations: one to add a finalizer that is executed when the scope is closed, and one to actually close the scope. A computation that acquires a resource requires that a `Scope` is in the environment. After the resource is acquired, a finalizer for releasing the resource is added to the scope. Before the ZIO is executed, the `Scope` must be provided. The provided `Scope` determines how long the resource is usable and when it is released.

To create a resource, ZIO has `acquireRelease` constructor and several variants for it. Like the name suggests, these constructors take two ZIO computations as

their parameters: one to acquire the resource and one to release it. They return a ZIO computation that succeeds with the resource, and has added `Scope` to the environment. In order to determine the extent of a `Scope` and remove it from the environment, ZIO provides an operator called `scoped`. This function takes a ZIO computation as an argument that requires a scope. It then opens the scope, runs the computation with the scope, and finally closes the scope. Several resourceful ZIOs could be interpreted in different ways depending on how the `Scope` is provided, which changes the order of how resources are acquired and released, in other words the lifetime of the resource. Listing 59 demonstrates use of these operators, and how scoping affects the order of acquiring and releasing resources.

Listing 59 Operators for acquiring resources and providing a `Scope` in ZIO. Resources can be scoped to shared or separate scopes.

```
def log(msg: String): UIO[Unit] = ZIO.debug(msg)

val intResource: ZIO[Scope, Nothing, Int] = ZIO.acquireRelease(
  acquire = log("acquire int").as(34),
)(release = int => log(s"release $int"))

val stringResource: ZIO[Scope, Nothing, String] = ZIO.acquireRelease(
  acquire = log("acquire string").as("foo"),
)(release = str => log(s"release $str"))

// "acquire int", "acquire string", "release foo", "release 34"
val program1 = ZIO.scoped { intResource *> stringResource }

// "acquire int", "release 34", "acquire string", "release foo"
val program2 = ZIO.scoped(intResource) *> ZIO.scoped(stringResource)
```

If multiple resources are acquired, they are released in the reverse order. By default releasing resources happens sequentially, but `Scope` also enables running finalizers in parallel if configured to do so. When using `Scope` with `ZIO.scoped`, finalizers are guaranteed to be executed even when an error/defect is encountered, or when the workflow is interrupted.

Traditionally resource management is implemented with a `try-finally` statement, where a resource is acquired before using it in a `try` block, and lastly releasing it in a `finally` block. This guarantees that the resource is released, even if an error occurs after acquiring the resource. Managing resources with `try-finally` lacks in expressivity, composability, and safety compared to a higher-level declarative strategy like `Scope`. Firstly, the acquired resource is not visible in the type system, so it is possible to forget to release the resource. Secondly, composing the acquisition and release of several resources with `try-finally` is complicated, especially if the acquisition and release must be done in a particular order. Thirdly, when a resource is acquired, the lifetime of the resource must be statically determined (by adding a `finally` statement).

4.5 Concurrency

ZIO values are descriptions of workflows that can be executed in different ways. They can be executed sequentially or concurrently, and this decision can be made after a ZIO workflow is defined. This makes ZIO, or any other IO monad, an ideal abstraction for high level combinators that allow the programmer to precisely define the concurrency semantics of a computation.

The concurrency model in ZIO is based on fibers. Every operation that waits for another ZIO fiber to complete is semantically blocking and does not block actual operating system threads. For code that is doing blocking IO, ZIO has a separate thread pool dedicated for blocking operations. ZIO uses structured concurrency by default and allows other types of concurrency semantics to be configured when needed. Additionally ZIO offers many concurrency primitives such as queues, atomic references and semaphores, as well as software transactional memory, which are not discussed in more depth in this thesis.

Every ZIO workflow is executed by a fiber that is in turn executed by the ZIO

runtime that assigns and schedules fibers to be run on actual threads. In ZIO, fiber is a datatype that is a handle to an ongoing computation. A ZIO program is started on a fiber called *main fiber* created by the runtime. Additional fibers can be created with the `fork` operator on a ZIO workflow. The `fork` operator starts executing the forked fiber concurrently in the background and then returns immediately to the original fiber. Other common operations with fibers are to check whether a fiber is finished (`poll`), to wait for the result (`join` and `await`), or to interrupt a fiber's execution (`interrupt`). Most operations on fibers are effects, and thus they return their result inside a ZIO. Listing 60 demonstrates forking and joining a fiber.

Listing 60 Forking and joining a fiber in ZIO.

```
val work = ZIO.sleep(1.second) *> ZIO.debug("Work completed")
val parentZIO = for
  childFiber <- work.fork
  _          <- ZIO.debug("Parent forked child fiber")
  _          <- childFiber.join
  _          <- ZIO.debug("Parent joined child fiber")
yield ()

// When executed prints:
// Parent forked child fiber
// Work completed
// Parent joined child fiber
```

Structured concurrency in ZIO is implemented with a fiber *supervision* model. Every forked fiber in ZIO has a scope that determines the maximum lifetime of a fiber. When a scope is closed, all fibers in that scope (that have not finished executing) are interrupted. The scope is determined at the time of forking, and it depends on which operator the forking is done with. It is also possible to change the scoping of a fiber after it is forked, but this is somewhat rare. Listing 61 introduces different forking operators and their type signatures.

The default is to scope child fibers to their parent, which is achieved with the

Listing 61 Forking operators on ZIO.

```
trait ZIO[-R, +E, +A]:
  def fork: URIO[R, Fiber[E, A]]
  def forkDaemon: URIO[R, Fiber[E, A]]
  def forkScoped: URIO[R & Scope, Fiber[E, A]]
  def forkIn(scope: Scope): URIO[R, Fiber[E, A]]
```

`fork` operator. In order for a fiber to outlive its parent, a different operator is required. If a fiber should live forever, independently from its parent, `forkDaemon` operator attaches the fiber to the *global scope*, which is closed only when the whole application exits. For finer-grained control over the scope of a fiber, its lifetime could be tied to ZIO `Scope`, with `forkScoped` operator, which is scoped to surrounding `Scope` in the ZIO environment, or `forkIn` operator, which takes a `Scope` as an argument. Listing 62 demonstrates forking fibers in different scopes and their interruption properties.

Listing 62 Fiber scopes and interruption in ZIO

```
def log(msg: String): UIO[Unit] = ZIO.debug(msg)
def hangForever(tag: String): UIO[Nothing] =
  log(s"Start: $tag") *> ZIO.never.onInterrupt(log(s"Stop: $tag"))

val supervision: UIO[Unit] = for
  _     <- hangForever("fork").fork
  _     <- hangForever("forkDaemon").forkDaemon
  scope <- Scope.make
  _     <- hangForever("forkIn").forkIn(scope)
  _     <- ZIO.scoped(hangForever("forkScoped").forkScoped)
  _     <- scope.close(Exit.unit)
yield ()

// Start order(non-deterministic): fork, forkDaemon, forkIn, forkScoped
// Interruption order: forkScoped, forkIn, fork
// forkDaemon is not interrupted
```

The fibers of an application can be thought of as a tree where the main fiber is the root node, new child nodes are created by a `fork` operation, and each parent

fiber is the root node of its subtree from which all child fibers branch. When a fiber terminates, either by succeeding, failing, or by interruption, all of its descendant fibers are recursively interrupted. After the child fibers have been interrupted, the current fiber's finalizers are executed. A call to interrupt a fiber blocks until the fiber has interrupted all of its children, and all finalizers have finished executing. If a fiber has a large number of descendants with long-running or many finalizers, the interruption could take a significant amount of time. Sometimes it is desired to perform the interruption in the background by a daemon fiber and return immediately to the fiber that initiated the interrupt. This can be achieved by interrupting the fiber with `interruptFork` method or by using `disconnect` combinator on a ZIO workflow to make the interruption happen in the background.

Sometimes a fiber is doing critical work, such as disposing acquired resources, that cannot be interrupted without leaving the program in an inconsistent state. These parts of a program should therefore be executed without interruptions. ZIO guarantees that if a fiber that is executing a section marked as uninterruptible is interrupted by another fiber, the uninterruptible section is executed to completion despite the interruption. A ZIO workflow can be marked as uninterruptible with `uninterruptible` and `uninterruptibleMask` operators. The former marks the whole ZIO workflow as uninterruptible, while the latter gives more control over what parts inside an uninterruptible section are interruptible.

Fibers along with other concurrency primitives are basic building blocks for creating concurrency operators in ZIO. Countless concurrent and parallel combinators can be implemented with forking, joining and interrupting fibers in various ways. Combinators implemented with fibers automatically inherit structured concurrency properties like supervision, scoping and interruption. Listing 63 demonstrates how the `zipPar` concurrency operator can be implemented using fibers.

Fibers are a low-level construct and programming directly with them is error-

Listing 63 `zipPar` implementation with fibers in ZIO.

```
// Actual implementation in ZIO is considerably more complex due to
// environment, errors, race conditions, and other concerns
def zipPar[A, B](left: UIO[A], right: UIO[B]): UIO[(A, B)] =
  for
    fiber1 <- left.fork
    fiber2 <- right.fork
    a      <- fiber1.join
    b      <- fiber2.join
  yield (a, b)
```

prone because of the possibility of race conditions. ZIO has numerous built-in high-level concurrency operators (a few of which are presented below) that should be used instead of fibers, when possible. Operators that combine multiple ZIOs in parallel are usually suffixed with `Par` to indicate that execution happens in parallel. For the majority of the operators that combine several independent ZIOs, there is a parallel counterpart that executes in parallel. Listings 40 and 41 demonstrate ZIO combinators that combine several ZIOs sequentially. Their parallel counterparts include `zipPar`, `foreachPar`, and `collectAllPar`, to name a few. Some operators only make sense to be defined as parallel, such as `race` and its variants that execute multiple ZIOs and pick the one that succeeds first.

Many combinator operators (like `foreach`, `collectAll`, and every `zip` variant) need the result of each combined ZIO in order to compute a result. Consequently, if even one of the combined ZIOs fail, the result cannot be computed. In sequential composition this is unproblematic: if a ZIO fails, the execution of subsequent ZIOs will not be started. When composing ZIOs in parallel, the semantics are more complicated. All composed ZIOs start executing in parallel and if any of them fails, the results of the others are not needed anymore and they are interrupted. In some situations this interrupting behavior is not desired, and it can be avoided by converting ZIOs to infallible, with operators described in Section 4.2, before the parallel compo-

sition. Listing 64 demonstrates the `zipPar` operator and an interruption associated with it.

Listing 64 Parallel composition of ZIOs with `zipPar` operator.

```
// Represents long-running interaction such as network or file system
def work(duration: Duration) = ZIO.sleep(duration)

val fast: IO[String, Int]  = work(50.millis) *> ZIO.fail("oops")
val slow: IO[Nothing, Int] = work(3.seconds) *> ZIO.succeed(34)

// 'slow' is interrupted after 50ms when 'fast' fails
val successInterrupted: IO[String, (Int, Int)] =
  fast.zipPar(slow)

// 'slow' is not interrupted because 'fast' is made infallible
val successNotInterrupted: IO[Nothing, (Either[String, Int], Int)] =
  fast.either.zipPar(slow)
```

By default parallel combinators in ZIO have unbounded parallelism, which means that all composed ZIOs are executed at the same time. Often one would want to limit the amount of parallelism, especially with operators like `foreachPar` or `collectAllPar`, whose parallelism is defined by the size of a collection received as an argument. ZIO has two basic operators for controlling the amount of parallelism: `withParallelism` that limits concurrency to a number it receives as an argument, and `withParallelismUnbounded` that removes any limitations to parallelism. These operators only apply to a single ZIO workflow, meaning that parallelism is limited only in a specific ZIO. Composing ZIOs with varying parallelism limits preserves the parallelism of each individual ZIO workflow. Listing 65 demonstrates the use of operators controlling the amount of parallelism.

Listing 65 ZIO operators for controlling the amount of parallelism.

```
def fetchContent(url: URL): IO[Throwable, String] = ???
val urls: List[URL]                                = ???

val contents: IO[Throwable, List[String]] =
  ZIO.foreachPar(urls)(fetchContent)

// By default all requests are performed in parallel
val unboundedParallelism = contents

// The parallelism is limited to 10 concurrent requests
val boundedParallelism =  unboundedParallelism.withParallelism(10)

// Bounded parallelism can be converted back to unbounded
val unboundedAgain = boundedParallelism.withParallelismUnbounded
```

4.6 Summary of ZIO

ZIO is a realization of the monadic approach to effectful programming. It puts into practice results of several decades of programming research. Essentially ZIO is an IO monad and it inherits much of their properties, both good and bad. On the positive side, effects can be described in a referentially transparent way, which gives high expressivity and good refactoring characteristics. On the negative side, like with all monads, encoding multiple effects is not straight-forward and programs must be written in a sometimes cumbersome monadic syntax.[1]

ZIO circumvents some of the problems traditionally related to monads, such as impaired type inference and having to encode multiple effects by nesting monads or by using monad transformers. The majority of the practically useful effects can be encoded by using a monad with three type parameters. First two parameters are used to include the capabilities of IO and Either monads. The third parameter is the environment that allows to encode other, possibly overlapping, effects such as Reader and State. The environment takes inspiration from algebraic effects and

[1]Direct syntax can be achieved by utilizing compiler macros that rewrite direct style to monadic style at compile time, e.g https://zio.dev/zio-direct/.

handlers, and is able to encode semantics similar to that approach. The environment is also central to dependency injection in ZIO.

ZIO takes a novel approach with its error model, which enables expressing errors in a referentially transparent way. The error model is expressive, capable of encoding asynchronous and concurrent errors that are in the core of modern applications. The error model blends well with the concurrency model, which enables the programmer to express concurrency concerns at a high abstraction level. When multiple effects are encoded in a single monad, IO, errors and concurrency compose together in a natural and type safe way.

ZIO provides many state-of-the-art features with the focus on practical usability. Although the library is quite new, it has few years of production experience from several large companies. It has a comprehensive and constantly growing ecosystem. Because ZIO is built with Scala, which is a JVM language, ZIO programs also have access to Java libraries, one of the largest open source ecosystems. For these reasons, ZIO is one of the most viable ways to develop modern applications today.

5 Case study

This chapter reports on a study focusing on the practical applicability of ZIO in a development of a server application for Qlik Sense business intelligence tool [73]. First the purpose and the background of the project is introduced along with a quick overview of Qlik Sense. Then, the evaluation of ZIO is discussed, divided into several sections, roughly following the structure of Chapter 4 about ZIO. The first section examines error handling with ZIO and its usability in the development process. The second section discusses the use of dependency injection with ZIO. The third section looks at testing and how ZIO facilitates the implementation of automated tests. The fourth section covers the role of ZIO's concurrency constructs in the development of the application. The last section analyzes the overall usability of ZIO in application development.

The application under study is the backend for a web application. It exposes its functionality via an HTTP interface. The purpose of the application is to manage control parameters of the Qlik Sense business intelligence tool. A browser-based user interface has also been developed for the application, but it is not within the scope of this thesis.

Qlik Sense is a data analytics software with Extract, Transform and Load (ETL), data modeling and interactive data visualization capabilities. Qlik Sense is commonly used to create dashboards that display data from various sources in a single easy-to-consume format. The user of the dashboard can filter the data and export

reports in various formats, such as PDF or Excel, or via email message. Qlik Sense has various deployment options including on-premises servers, Kubernetes and more recently also a Cloud/SaaS offering.

It is often desirable to parameterize the operation of Qlik Sense applications, such as URLs, calculation formulas and titles to display. However, Qlik does not have a built-in mechanism to maintain such configurations, and many Qlik applications end up managing configurations in an Excel file or similar. This method is often perceived as suboptimal due to usability challenges for non-technical users, limited possibilities to manage user rights, and deficiencies in validating the correct structure of the configuration, which makes it error prone. A table as a configuration format is also limited in describing, for example, object or array structures.

For these reasons, a custom web application for managing Qlik Sense configuration was developed. The custom solution exposes an HTTP/JSON API that Qlik Sense applications can read configuration values from. Authentication is implemented with an API key, which is simply a token provided by the client in the HTTP headers. The configuration format and values can be managed from a graphical user interface by Qlik application developers.

The code of the application was divided roughly into three layers/modules: core, persistence and HTTP. The core contains all the business logic and interface definitions that are implemented in the persistence layer. The HTTP layer is responsible for exposing the public API of the application. This requires decoding HTTP requests and handling authentication logic. The persistence layer implements interfaces defined by the core and HTTP layers. It translates domain models into relational format and communicates with a PostgreSQL database, which is used for persistence.

The application utilizes many libraries from the ZIO ecosystem: zio-protoquill for database interaction, tapir with zio-http for the HTTP server, zio-json for JSON

(de)serialization, zio-config for reading and parsing the application configuration, zio-logging for logging and zio-test with zio-testcontainers for testing. The server application is packaged as a Docker container to accommodate the different deployment options of Qlik Sense.

5.1 Error handling

The ZIO error model was proven to be suitable for many situations. Due to the greenfield nature of the project, the development process involved a significant amount of experimentation and refactoring. Changes could be done with confidence because of statically typed errors that trigger a compile error if an error case was accidentally left unhandled. Adding a new error case to a method is easy because the compiler errors indicate where additional error handling is required.

Converting errors between different layers of the application turned out to be easy and convenient. Listing 66 contains part of the `ApikeyRepository` implementation that stores API keys in a PostgreSQL database. Because every API key value must be unique, adding a new API key can fail if the database already contains an API key with the same value. The possibility of failure is reflected in the return type of the `add` method: `IO[DuplicateApikey, Unit]`. Running an insert query against the database can fail with `SQLException`, which must be converted to `DuplicateApikey` or to a ZIO defect, depending on the specific `SQLException` received. The `catchAll` operator expresses this logic clearly.

Retrying capabilities of ZIO also proved to be useful. The data model for API key contains a secret token and a description. In the application `ApikeyService` is responsible of creating a new API key. The user can specify a description for the key and `ApikeyService` is responsible of creating a token and persisting the new API key. The process of creating a new API key consists of validating that the provided

Listing 66 Expressing the desired error handling behavior with `catchAll` operator.

```
class PostgresApikeyRepository(datasource: DataSource):
  def add(apikey: Apikey): IO[DuplicateApikey, Apikey] =
    // Prepare the insert query, implementation omitted for brevity
    val insertApikeyQuery: Query[Apikey] = ???

    // Run the query against a database
    val queryResult: IO[SQLException, Unit] = run(insertApikeyQuery)

    // Convert the SQLException to DuplicateApikey error or defect
    val uniqueViolationHandled: IO[DuplicateApikey, Unit] =
      queryResult
        .catchAll {
          case exc: SQLException if isUniqueViolation(exc) =>
            ZIO.fail(DuplicateApikey(apikey))

          case otherSqlExc: SQLException => ZIO.die(otherSqlExc)
        }

    // Return the original apikey after insert was successful
    uniqueViolationHandled.as(apikey)

  private def isUniqueViolation(exc: SQLException): Boolean =
    exc.getSQLState == PSQLState.UNIQUE_VIOLATION.getState
```

description meets its requirements, generating a new token, persisting the new API key and finally returning the created and persisted API key. The service delegates the creation of the token to `KeyGenerator` and persistence to `ApikeyRepository`.

Listing 67 Expressing sophisticated retry policies declaratively with `Schedule` and `retry` operator on ZIO.

```
class ApikeyService(keyGen: KeyGen, repo: ApikeyRepository):
  def create(description: String): IO[InvalidDescription, Apikey] =
    type CreateError = DuplicateApikey | InvalidDescription

    val createApikey: IO[CreateError, Apikey] = for
      validDesc   <- validateDesc(description)
      token       <- keyGen.generateKey
      savedApikey <- repo.add(Apikey(validDesc, token))
    yield savedApikey

    // Retry only in the case of DuplicateApikey and at most 10 times
    val policy = Schedule.recurWhile[CreateError] {
      case DuplicateApikey(_)    => true
      case InvalidDescription(_) => false
    } && Schedule.recurs(10)

    // Apply the retry policy to the ZIO that creates the apikey
    val retried: IO[CreateError, Apikey] = createApikey.retry(policy)

    // Change the error type of the retried ZIO:
    // IO[CreateError, Apikey] => IO[InvalidDescription, Apikey]
    // By converting all errors to defects except InvalidDescription
    retried.refineOrDieWith {
      case descError: InvalidDescription => descError
    }(otherErr => RuntimeException(s"Defect in create: $otherErr"))

  def validateDesc(str: String): IO[InvalidDescription, String] = ???
```

As demonstrated in Listing 66, persisting the API key can fail if the token already exists in the database. A desired way to react to this situation is by creating a new token and trying to persist the API key again with the new token. However, it is desirable not to retry persisting a new API key indefinetely. If persisting a new API key fails a couple times in a row with `DuplicateApikey`, there is prob-

ably a bug in the code, and it should be considered a defect. Listing 67 shows the `ApikeyService.create` method that implements the above logic with ZIO. A ZIO `Schedule` describes the retry policy and `refineOrDieWith` is used to convert `DuplicateApikey` to defect if retries do not resolve the error.

5.2 Dependency injection

The services in the application were implemented by using constructor-based dependency injection. If a service requires another service, it will receive the dependency as a constructor argument. This pattern can be seen in Listings 66 and 67 where dependencies are received as constructor arguments. Each service defines a `ZLayer` in its companion object, which can be used to construct that specific service. Dependencies for the program are provided in the main method by referencing `ZLayer` of each required service. Listing 68 demonstrates how the layers are provided.

ZIO resolves and constructs the dependency graph at compile time, as described in Section 4.3.1. If a required dependency is not provided, ZIO reports a developer-friendly error message explaining what dependency is missing. This compile time verification proved to be valuable in the development process when new dependencies were added to services. Forgetting to provide a newly-added dependency was brought to the attention of the developer in a clear format before the application could even be started. This can be demonstrated, for example, by commenting out `KeyGen.layer` and `Database.dataSourceLayer` from Listing 68. The compiler reports an error shown in Figure 5.1, which clearly states what dependencies are missing and which services require them.

Listing 68 Dependencies are provided in the main method of the application as ZLayers.

```
object Main:
  // Represents the whole program before its dependencies are provided
  val program: ZIO[ApikeyService, Nothing, Unit] = ???

  val run: ZIO[Any, Nothing, Unit] = program.provide(
    ApikeyService.layer,
    PostgresApikeyRepository.layer,
    KeyGen.layer,
    Database.dataSourceLayer,
  )
```

```
[error] 12 |  program.provide(
[error]    |  ^
[error]    |
[error]    |
[error]    |  ——— ZLAYER ERROR ———————————————————————————————————————————
[error]    |
[error]    |   Please provide layers for the following 2 types:
[error]    |
[error]    |     Required by ApikeyService.layer
[error]    |     1. example.KeyGen
[error]    |
[error]    |     Required by PostgresApikeyRepository.layer,
[error]    |     2. javax.sql.DataSource
[error]    |
[error]    |  ————————————————————————————————————————————————————————————
[error]    |
[error] 13 |    ApikeyService.layer,
[error] 14 |    PostgresApikeyRepository.layer,
[error] 15 |    // KeyGen.layer,
[error] 16 |    // Database.dataSourceLayer,
[error] 17 |  )
[error] one error found
[error] (Compile / compileIncremental) Compilation failed
```

Figure 5.1: Error message produced by ZIO when required `ZLayer` is not provided.

5.3 Testing

The application contains several automated tests. ZIO has its own test library, zio-test, which was used for testing. Three types of tests were written for the application: unit tests, integration tests and system tests. Unit tests run in-memory and exercise a single class or function. Integration tests ensure the correct functionality of multiple services together and may include out-of-process dependencies such as

databases. Repository classes that interact with a PostgreSQL database were tested with integration tests that used a database instance running in a Docker container. In system tests the application is tested as a whole, which in this case means that the configuration is read from environment variables and the database is running in a Docker container, similar to integration tests. System tests treat the application as a black box and tests only interact with its public API, which in this case consists of the HTTP endpoints.

Dependency injection in zio-test is managed with `ZLayer`s. This makes it easy to configure tests in a way that the class under test can be provided with fake implementations of its dependencies. ZIO has also built-in *test services* that make it possible to write deterministic tests that interact with the console, time/clock, random number generator and environment variables. This ability to effortlessly test interaction with time and environment variables in a controlled manner proved to be valuable.

Listing 69 shows a test for `ApikeyService` that verifies that the revokation time of an API key is set to the current time. In the test a `TestClock` is used to fix the current time visible to the service, and then assert that the hardcoded time was actually used. The example also demonstrates how layers are used to provide dependencies to the class under test.

The biggest advantage of ZIO test services were the possibility to configure the environment variables in system tests. The application reads its configuration, such as a database connection string and an HTTP port number, from environment variables. Traditionally testing how a program interacts with environment variables is cumbersome and error prone to say the least. ZIO `TestSystem` allows setting environment variables easily before the application is started and it tries to read its configuration. Listing 70 shows a layer that requires environment variables that will be set before the application is started and kept running in the background.

Listing 69 ZIO `TestClock` facilitates testing of code that uses the current time. `ZLayers` enable to inject desired dependencies to the service under test.

```
test("revoke should set current time as the revokation time") {
  val fixedTime: Instant = Instant.parse("2023-03-30T19:34:28Z")

  for
    apikeyService <- ZIO.service[ApikeyService]
    apikey        <- apikeyService.create("test apikey description")

    _ <- TestClock.setTime(fixedTime) // Set current time
    _ <- apikeyService.revoke(apikey) // Perform logic under test

    // This is verifiable using the provided in-memory repository
    allApikeys    <- FakeApikeyRepository.getAll
    revokedApikey <- ZIO.getOrFail(allApikeys.find(_ == apikey))

  // Assert that the fixed time was used as the revokation time
  yield assertTrue(revokedApikey.isRevokedAt(fixedTime))
}.provide(
  ApikeyService.layer,
  FakeApikeyRepository.layer,
  FakeKeyGen.layer,
) // Test is configured with real ApikeyService and fake dependencies
```

Listing 70 ZIO `TestSystem` makes it possible to set/overwrite environment variables the application sees. This is used to set the configuration for the application in system tests.

```
case class EnvVars(values: Map[String, String])

object SystemTestSetup:
  // This layer starts the application in the background and
  // configures it by setting environment variables before starting.
  val layer: ZLayer[EnvVars, Nothing, Unit] =
    TestSystem.default >>> ZLayer.scoped {
      for
        envVars <- ZIO.service[EnvVars]
        _       <- setEnvironmentVariables(envVars)
        _       <- startApp
      yield ()
    }

  // The application keeps it running while tests are finished.
  // Delay makes sure the application has had time to start.
  def startApp = Main.run.forkScoped *> ZIO.sleep(2.second)

  def setEnvironmentVariables(envVars: EnvVars) =
    ZIO.foreachDiscard(envVars.values) { (key, value) =>
      TestSystem.putEnv(key, value)
    }
```

ZIO test services do not provide a way to control access to the file system, which is also quite hard to test, for similar reasons as environment variables. In its current form, the application does not use the file system. If it did, it is evident that the application programmer would like ZIO to provide tools to test such interactions.

5.4 Concurrency

The application in its current form is quite simple, and thus ZIO's concurrency features were not needed much. This is partly due to the decision to use a relational database, which can perform complex query logic in a single query. In other projects of similar complexity, the need for concurrency typically arises when fetching data

from multiple sources and combining the data in the application code. This occurs, e.g, with NoSQL databases, which are usually not capable of doing joins in the database, thus forcing the joining logic to be expressed in the application code.

As the application advances, we foresee that concurrency plays a more significant role. ZIO's concurrency features will likely then become more useful, especially when the application gains other functionalities besides an HTTP interface. Useful concurrency features could include scheduled batch jobs running in the background, such as updating and invalidating caches or reporting metrics, or asynchronous messaging where the application must listen and react to new messages in the background. ZIO's concurrency and scheduling capabilities are well suited for these kind of use cases.

5.5 Analysis

The case study revealed that using ZIO may initially slow down development, but this is only temporary and lasts for days or about one week. While the simplest tasks may sometimes be slightly more challenging to implement with ZIO, the benefits become apparent when dealing with more complex problems. The use of ZIO made it easier to tackle more difficult problems that may typically be ignored in imperative languages due to the time and effort required to solve them, such as examples in Listings 66 and 67 demonstrate. Some tasks that are unreasonably difficult (or even practically impossible) in mainstream languages are possible, often simple, with ZIO, such as determining dynamic lifetime for a resource with `Scope` or defining a complex retry policy with `Schedule`.

Acknowledging the ever changing and unpredictable nature of software projects, building new projects on strong foundations is desirable. This increases the likelihood that customizing and evolving the software is possible with reasonable effort. In our small experiment, ZIO proved to be a robust foundation that enables such

customizations. Refactoring was easy and could be done with confidence because ZIO programs are referentially transparent.

Developers with no previous experience with monadic effects or effects as values may find it difficult to comprehend programs written with ZIO. This became clear in discussions with other developers involved in the project. The programmer must master quite a bit of functional programming techniques and adopt a functional programming mindset in order to use ZIO (or other monadic effects) effectively.

One of the key benefits of monads is their ability to describe different aspects of the control flow in a clear and modular manner. Monads facilitate good separation of concerns by allowing developers to define the core business logic, often referred to as "happy path", separately from error handling and concurrency concerns. This separation of concerns facilitates writing maintainable and scalable code. In contrast, the imperative paradigm often conflates these concerns, forcing programmers to deal with multiple aspects in a single block of program logic.

6 Conclusion

Monads as a way to encode effects were discovered in the 90s. It is possible to use monads in a majority of current languages, as long as the language has support for higher-order functions. Assuming a statically typed language, monads also provide an effect system, in addition to modeling effects. Even though monads are not part of mainstream industrial programming, they have been used for a long time and they are quite a mature approach today. Challenges with monads is that they force programs to be written in a rather cumbersome monadic syntax. Also, combining different monadic effects is not straightforward and necessitates the use of complex programming constructs.

Algebraic effects and handlers are a more recent approach for managing effects that was discovered in the early 2000s; first academic languages appeared in the 2010s. First languages with support for algebraic effects intended for commercial use surfaced in the early 2020s. A productive use of algebraic effects requires a language that has native support for them. Such languages usually come with a built-in effect system as well. These languages allow programmers to write effectful programs in direct style, and combining different effects is effortless. Algebraic effects and handlers are, however, a recent practice with many remaining open questions regarding how they should be best included in a programming language.

Programming in a direct style with algebraic effects resembles imperative programming. It can be argued that the direct style of programming is more familiar

to the majority of programmers, thus making algebraic effects easier to comprehend than monadic effects. Monadic effects, on the other hand, are far more accessible to the average programmer than algebraic effects, since there are several monadic effect libraries available for different languages. Both approaches enable highly expressive and modular effects; monads with combinators that modify a value representing a computation and algebraic effects with handlers that interpret the effect in a specific manner.

Capability based effects address many shortcomings of monads and algebraic effects. Their research is ongoing, and it is not yet possible to use them in practical applications, since languages with support for them are experimental research languages. Nevertheless, proposals related to effect polymorphism seem to go a long way to make capability based effects more practical and easier to use than other sophisticated approaches for managing side effects.

Compared to unrestricted side effects, monads and algebraic effects provide attractive ways to manage side effects. Regarding the research questions formulated in the introduction, it can be concluded that controlling side effects with monads and algebraic effects is clearly more expressive and compositional than unrestricted side effects. This is underlined by how much more convenient it is to implement re-usable logic for effects, such as retries and timeouts, with monads and algebraic effects than it is with unrestricted side effects.

Programs written with monadic or algebraic effects have a tendency to be more declarative than their imperative counterparts with unrestricted side effects. These features facilitate the implementation of modular and resilient programs that are easier to modify and that respond to errors in a clearly defined manner. Concurrency concerns can be alleviated: high-level concurrency makes it easier to implement correct and performant programs when compared to working with traditional imperative low-level primitives, such as threads.

The case study described in this thesis showed that ZIO provides the programmer a good foundation for managing control flows and abstractions, encountered for example in error handling, and leads to declarative and concise programs. Expressing programs in referentially transparent way proved to be beneficial for refactoring, which encourages changing and correcting the program structure as the application evolves. ZIO's benefits are fully realized when approaching problem-solving from a functional programming perspective, which can also pose a weakness: the advantages it provides may not be immediately apparent to programmers who are exclusively familiar with imperative languages.

Algebraic effects with handlers and capability based solutions may eventually turn out to provide better developer ergonomics compared to monads, but currently there is little to none practical experience of using them in commercial software. It remains to be seen whether this sophisticated approach for managing side effects will make their breakthrough in the industry. Eventually it is a trade-off; is a more sophisticated approach perceived useful enough to justify the initial effort of education/learning that it requires. In turn, is it possible to make these more sophisticated approaches more accessible by making them feel more familiar to the practicing programmers, thus requiring less training? In the meantime, ZIO may well be one of the most compelling technologies to try out to get a taste of what these more advanced approaches of handling side effects can offer today.

References

[1] "TIOBE index, January 2022". (2022), [Online]. Available: https://www.tiobe.com/tiobe-index (visited on 01/13/2022).

[2] "Stack Overflow developer survey 2022". (2022), [Online]. Available: https://survey.stackoverflow.co/2022 (visited on 03/24/2023).

[3] H. Abelson and G. J. Sussman, *Structure and interpretation of computer programs*, 2nd ed. The MIT Press, 1996.

[4] R. Cartwright and M. Felleisen, "Extensible denotational language specifications", in *Theoretical Aspects of Computer Software: International Symposium TACS'94 Sendai, Japan, April 19–22, 1994 Proceedings*, Springer, 1994, pp. 244–272.

[5] S. L. Peyton Jones and P. Wadler, "Imperative functional programming", in *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993, pp. 71–84.

[6] S. Lindley, C. McBride, and C. McLaughlin, "Do be do be do", Association for Computing Machinery, 2017, pp. 500–514. DOI: 10.1145/3009837.3009897. [Online]. Available: https://doi.org/10.1145/3009837.3009897.

[7] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow, "A semantics for imprecise exceptions", in *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, 1999, pp. 25–36.

[8] S. P. Jones, "Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell", *NATO Science Series Sub-Series III Computer and Systems Sciences*, vol. 180, pp. 47–96, 2001.

[9] S. Marlow, S. P. Jones, A. Moran, and J. Reppy, "Asynchronous exceptions in Haskell", in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 274–285.

[10] D. Yuan, Y. Luo, X. Zhuang, *et al.*, "Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, USENIX Association, 2014, pp. 249–265.

[11] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms", in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 267–275.

[12] "Structured concurrency". (2016), [Online]. Available: `https://250bpm.com/blog:71` (visited on 02/22/2023).

[13] "Notes on structured concurrency, or: Go statement considered harmful". (2018), [Online]. Available: `https://vorpus.org/blog/notes-on-`

structured-concurrency-or-go-statement-considered-harmful/
(visited on 02/22/2023).

[14] "Kotlin coroutines v0.26.0: Structured concurrency". (2022),
[Online]. Available:
https://github.com/Kotlin/kotlinx.coroutines/releases/tag/0.26.0
(visited on 12/02/2022).

[15] "Swift structured concurrency". (2022),
[Online]. Available: https://github.com/apple/swift-evolution/blob/main/proposals/0304-structured-concurrency.md
(visited on 12/02/2022).

[16] "Java project loom". (2022), [Online]. Available:
https://openjdk.org/jeps/428 (visited on 12/02/2022).

[17] "The Scala programming language". (2022),
[Online]. Available: https://www.scala-lang.org/ (visited on 01/13/2022).

[18] B. C. Oliveira, A. Moors, and M. Odersky, "Type classes as objects and implicits", *ACM Sigplan Notices*, vol. 45, no. 10, pp. 341–360, 2010.

[19] "Unison programming language". (2022), [Online]. Available:
https://www.unison-lang.org/ (visited on 09/06/2022).

[20] "Koka programming language". (2022),
[Online]. Available: https://koka-lang.github.io/koka/doc/index.html
(visited on 09/06/2022).

[21] J. H. Morris, "Real programming in functional languages",
Xerox Palo Alto Research Center, Tech. Rep., 1981.

[22] D. K. Gifford and J. M. Lucassen,
"Integrating functional and imperative programming", in *Proceedings of the*

1986 ACM Conference on LISP and Functional Programming, 1986, pp. 28–38.

[23] "Frank programming language". (2022), [Online]. Available: `https://github.com/frank-lang` (visited on 09/06/2022).

[24] "OCaml programming language". (2022), [Online]. Available: `https://ocaml.org/` (visited on 09/06/2022).

[25] M. Odersky, A. Boruch-Gruszecki, E. Lee, J. Brachthäuser, and O. Lhoták, "Scoped capabilities for polymorphic effects", *ArXiv*, vol. abs/2207.03402, 2022.

[26] D. Leijen, "Type directed compilation of row-typed algebraic effects", in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 486–499.

[27] E. Moggi, "Computational lambda-calculus and monads", in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, USA: IEEE Press, 1989, pp. 14–23, ISBN: 0818619546.

[28] P. Wadler, "Comprehending monads", in *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, 1990, pp. 61–78.

[29] E. Moggi, "Notions of computation and monads", *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[30] P. Wadler, "Monads for functional programming", in *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, Springer, 1995, pp. 24–52.

[31] "Mozilla Developer Network: Array". (2021),
[Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array (visited on 07/15/2022).

[32] M. P. Jones,
"Functional programming with overloading and higher-order polymorphism", in *Advanced Functional Programming: First International Spring School on Advanced Functional Programming Techniques Båstad, Sweden, May 24–30, 1995 Tutorial Text 1*, Springer, 1995, pp. 97–136.

[33] "ZIO". (), [Online]. Available: https://zio.dev/ (visited on 08/18/2022).

[34] "Cats effect". (2022), [Online]. Available:
https://github.com/typelevel/cats-effect (visited on 08/18/2022).

[35] "Monix". (), [Online]. Available: https://github.com/monix/monix (visited on 08/18/2022).

[36] "Effect TS". (), [Online]. Available: https://github.com/Effect-TS/core (visited on 08/18/2022).

[37] "Arrow fx". (), [Online]. Available: https://github.com/arrow-kt/arrow (visited on 08/18/2022).

[38] "Missionary". (), [Online]. Available:
https://github.com/leonoel/missionary (visited on 08/18/2022).

[39] "Eff". (),
[Online]. Available: https://github.com/purescript/purescript-effect (visited on 08/18/2022).

[40] "Aff". (), [Online]. Available:
https://github.com/purescript-contrib/purescript-aff (visited on 08/18/2022).

[41] "Haskell do-notation". (), [Online]. Available: https://en.wikibooks.org/wiki/Haskell/do_notation (visited on 08/23/2022).

[42] "Scala for comprehensions". (), [Online]. Available: https://docs.scala-lang.org/tour/for-comprehensions.html (visited on 08/23/2022).

[43] "F# computation expressions". (), [Online]. Available: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions (visited on 08/23/2022).

[44] "OCaml binding operators". (), [Online]. Available: https://ocaml.org/manual/bindingops.html (visited on 08/23/2022).

[45] A. Filinski, "Representing monads", in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1994, pp. 446–457.

[46] J. I. Brachthäuser, A. S. Boruch-Gruszecki, and M. Odersky, "Representing monads with capabilities", in *HOPE 2021 Workshop*, 2021.

[47] "Monadic reflection". (2022), [Online]. Available: https://github.com/lampepfl/monadic-reflection (visited on 10/07/2022).

[48] P. Chiusano and R. Bjarnason, *Functional programming in Scala*, 2nd ed. Manning, 2014.

[49] G. Plotkin and J. Power, "Adequacy for algebraic effects", in *Foundations of Software Science and Computation Structures: 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings 4*, Springer, 2001, pp. 1–24.

[50] G. Plotkin and J. Power, "Computational effects and operations: An overview",
Electronic Notes in Theoretical Computer Science, vol. 73, pp. 149–163, 2004. DOI: https://doi.org/10.1016/j.entcs.2004.08.008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104050893.

[51] G. Plotkin and J. Power, "Algebraic operations and generic effects",
Applied categorical structures, vol. 11, pp. 69–94, 2003.

[52] G. Plotkin and M. Pretnar, "Handlers of algebraic effects",
in *Programming Languages and Systems: 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 18*, Springer, 2009, pp. 80–94.

[53] G. Plotkin and M. Pretnar, "Handling algebraic effects",
Logical Methods in Computer Science, vol. 9, no. 4, 2013.

[54] D. Leijen, "Algebraic effects for functional programming",
Technical Report. MSR-TR-2016-29. Microsoft Research technical report, Tech. Rep., 2016.

[55] "Idris effects". (2022), [Online]. Available: http://docs.idris-lang.org/en/latest/effects/index.html (visited on 09/21/2022).

[56] "Extensible effects". (2022), [Online]. Available: https://github.com/suhailshergill/extensible-effects (visited on 09/21/2022).

[57] "Algeff". (2022), [Online]. Available: https://github.com/brianberns/AlgEff (visited on 09/21/2022).

[58] "Eff programming language". (2022),
[Online]. Available: `https://www.eff-lang.org/` (visited on 09/06/2022).

[59] "Links programming language". (2022),
[Online]. Available: `https://links-lang.org/` (visited on 09/21/2022).

[60] "Effekt programming language". (2022),
[Online]. Available: `https://effekt-lang.org/` (visited on 09/21/2022).

[61] "Release of OCaml 5.0.0". (2022), [Online]. Available:
`https://ocaml.org/news/ocaml-5.0` (visited on 01/24/2023).

[62] M. Pretnar, "An introduction to algebraic effects and handlers. invited tutorial paper",
Electronic notes in theoretical computer science, vol. 319, pp. 19–35, 2015.

[63] J. I. Brachthäuser, P. Schuster, and K. Ostermann, "Effects as capabilities: Effect handlers and lightweight effect polymorphism", *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[64] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki, "The essence of dependent object types",
A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 249–272, 2016.

[65] "Capture checking". (2023), [Online]. Available: `https://docs.scala-lang.org/scala3/reference/experimental/cc.html` (visited on 02/13/2023).

[66] "What color is your function?" (2015), [Online]. Available:
`https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/` (visited on 11/29/2022).

[67] "Martin Odersky on Twitter: Caprese project". (2022), [Online]. Available: http://web.archive.org/web/20220711175044/https://twitter.com/odersky/status/1546552401368334339 (visited on 11/29/2022).

[68] "Rust documentation: Lifetimes". (2022), [Online]. Available: https://doc.rust-lang.org/rust-by-example/scope/lifetime.html (visited on 11/29/2022).

[69] S. N. S. Foundation. "Capabilities for typing resources and effects". (2022), [Online]. Available: https://data.snf.ch/grants/grant/209506 (visited on 11/28/2022).

[70] J. De Goes and A. Fraser, *Zionomicon*. Ziverge Inc., 2022.

[71] M. Snyder and P. Alexander, "Monad factory: Type-indexed monads", in *Trends in Functional Programming: 11th International Symposium, TFP 2010, Norman, OK, USA, May 17-19, 2010. Revised Selected Papers 11*, Springer, 2011, pp. 198–213.

[72] R. Hinze *et al.*, "Fun with phantom types", *The fun of programming*, pp. 245–262, 2003.

[73] "Qlik sense". (2023), [Online]. Available: https://www.qlik.com/us/products/qlik-sense (visited on 04/20/2023).

[74] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc", in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 60–76.

Appendix A Type classes

Type classes are a method to achieve ad-hoc polymorphism. They were first introduced by Wadler and Blott [74] in 1989 as a way to enable operator overloading in a programming language with Hindley-Milner type system. Eventually type classes were implemented in Haskell, largely based on Wadler's and Blott's proposal. A couple of years later, when applications of monads and related algebraic structures in programming were discovered [27], type classes were a natural way to implement such algebraic structures.

Type class is an abstraction that defines the behavior of a class of types. This enables one to implement generic functions that work with any type that is constrained to belong to a type class. A type is said to belong to a type class if it has an *instance* of that type class. An instance of a type class contains functions and/or values that implement the behavior of the type class. Instances are defined separately from the type for which the instance is. It is thus possible to provide type class instances for third party data types after they have been defined. The example in Listing 71 demonstrates how type classes enable the implementation of polymorphic functions that work with any data type that belongs to a specific type class.

Listing 71 Definition, implementation and use of the Ordering type class in Scala.

```
case class Money(amount: Int)

trait Ordering[A]:
  extension (lhs: A) def <(rhs: A): Boolean

given Ordering[Money] with
  extension (self: Money) def <(that: Money): Boolean =
    self.amount < that.amount

// Compatible with any data type that has instance for Ordering
def sort[A: Ordering](as: List[A]): List[A] = as.sortWith(_ < _)

val sorted = sort(List(Money(3), Money(1), Money(2)))
```