Design and Implementation of a Type-safe and Highly Concurrent Runtime System in F#

Daniel Larsen

June 18, 2022

Kongens Lyngby

DTU Compute
Department of Applied Mathematics and
Computer Science

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 45 25 30 31
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

In the current day and age, transistors have reached a nanoscopic level that makes it difficult for scientists and engineers to make them smaller. This causes chip manufacturers to improve performance of their chips by adding more independent processing cores, thus making the availability and ease of use of concurrent programming more important than ever before.

This dissertation addresses the design and implementation of a type-safe and highly concurrent runtime system called FIO written in the F# programming language. The purpose of FIO is to provide a developer-friendly API for the domain of concurrent programming together with an efficient runtime system that is capable of scheduling thousands of green threads simultaneously.

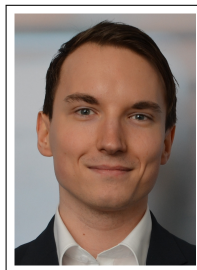
The purpose of this thesis is to design, implement and evaluate in terms of performance and scalability the above runtime system to determine whether it is feasible to achieve these goals using F#.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring the M.Sc. Eng. degree in Computer Science and Engineering.

The thesis deals with the design, implementation and evaluation of a type-safe and highly concurrent runtime system written in the F# programming language.

Lyngby, June 18 2022



Daniel Larsen

Daniel

Acknowledgements

Alceste Scalas, Associate Professor, DTU Compute

First and foremost, I would like to express my deepest appreciation for my thesis advisor, Alceste Scalas, for agreeing to have numerous meetings during the project and for always being ready with invaluable advice and suggestions when needed. Thank you very much for your support and understanding over these past months.

Thomas Mascagni, Thesis reviewer

I would like to thank my friend, colleague, and former fellow student, Thomas Mascagni, for helping reviewing the thesis, correcting mistakes and improving its overall quality.

Friends & family

At last, I would like to acknowledge and express my gratitude for the support and patience I have received from friends and family during this period.

Contents

Abstract **i**

Preface **iii**

Acknowledgements **v**

1 Introduction **1**
1.1 Objectives 3
1.2 Outline 3

2 A Quick Tour of FIO **5**

3 Background **13**
3.1 Functional programming theory 14
3.1.1 Pure functional programming 14
3.1.2 Functional effects 15
3.1.3 The IO monad 15
3.2 Domain-specific languages 17
3.2.1 Domain-specific language for an interpreted virtual machine 18
3.3 Functional programming toolkits for concurrent applications . . . 19
3.3.1 Cats Effect 19
3.3.2 ZIO 20
3.3.3 FIO (DC) 21
3.4 Summary 21

4 Design **23**
4.1 Effect system 24
4.1.1 Effect API 24
4.1.2 Effect type 27

4.1.3 Interpreter structure . . . 27
4.1.3.1 Visitor pattern . . . 29
4.1.3.2 Algebraic data types with type casting . . . 31
4.1.4 Effect structure . . . 32
4.1.4.1 Channels . . . 33
4.1.4.2 Fibers . . . 33
4.1.4.3 Opcodes . . . 34
4.2 Runtime system . . . 38
4.2.1 Naive interpreter . . . 39
4.2.2 Evaluation worker . . . 39
4.2.3 Blocking worker . . . 41
4.2.4 Intermediate interpreter . . . 41
4.2.5 Advanced interpreter . . . 42
4.2.6 Debugging tools . . . 43
4.3 Summary . . . 43

5 Implementation **45**
5.1 Effect system . . . 46
5.1.1 Effect API . . . 46
5.1.2 Interpreter structure . . . 48
5.1.3 Channels . . . 50
5.1.4 Fibers . . . 52
5.2 Runtime system . . . 54
5.2.1 Naive interpreter . . . 54
5.2.2 Intermediate interpreter . . . 57
5.2.2.1 Evaluation worker . . . 61
5.2.2.2 Blocking worker . . . 63
5.2.3 Advanced interpreter . . . 65
5.2.3.1 Evaluation worker . . . 65
5.2.3.2 Blocking worker . . . 67
5.2.4 Debugging tools . . . 67
5.2.4.1 Data structure monitor . . . 68
5.2.4.2 Deadlock detector . . . 70
5.3 Summary . . . 73

6 Evaluation **75**
6.1 Methodology . . . 76
6.2 Hardware specifications . . . 77
6.3 High precision timing . . . 77
6.4 Benchmark suite . . . 78
6.4.1 Pingpong . . . 78
6.4.2 Threadring . . . 79
6.4.3 Big . . . 80
6.4.4 Bang . . . 80

6.4.5 Spawn . . . 81
6.5 Results . . . 81
6.5.1 Pingpong . . . 82
6.5.2 Threadring . . . 83
6.5.3 Big . . . 85
6.5.4 Bang . . . 87
6.5.5 Spawn . . . 89
6.6 Summary . . . 91

7 Future Work **93**

8 Conclusion **95**

A API Function Implementation **97**

Glossary **101**

Abbreviations **103**

Bibliography **105**

Chapter 1

Introduction

In recent times, it has gotten increasingly difficult to improve the performance of modern CPUs due to the physical limitations of transistors. This can be understood through Moore's Law, which has been holding truth for more than 50 years, but is slowly coming to an end. Due to this, chip manufacturers like Intel and AMD have started to increase the number of independent cores in their CPUs as an alternative way to improve performance. [Cho22]

As a consequence, a paradigm shift in the way that scientists and engineers program CPUs is taking place. As the number of cores increase, so does the necessity for concurrent programming. Concurrent programming has a long history of being infamous for being less developer-friendly and more error-prone than sequential programming. This is primarily due to requiring the use of complex concepts such as shared state, synchronization, threads and more. Popular programming toolkits like Cats Effect and ZIO for Scala solves this problem by using functional programming theory and concepts such as the IO monad to abstract away details from the developer, and thereby making the usage of concurrent programming less complicated. For the F# programming language, no such framework or library exist that compares entirely to Cats Effect or ZIO. Seemingly, only a single attempt has been made – a library with the name of FIO by Daniel Chambers, which will be referred to as "FIO (DC)" throughout the thesis. [Cha19]

The aim of this thesis is to explore the feasibility of designing and implementing a type-safe and highly concurrent runtime system similar to ZIO in F# with the name of FIO. More precisely, the goal is to develop a type-safe and developer-friendly API for concurrent programming that will help ease the ongoing paradigm shift. The goal is for the runtime system to show high scalability in regards to concurrency by using green threads that are more efficient compared to system threads.

F# is chosen as the development language as it is a functional language and seemingly only a single attempt has been made at creating a type-safe and highly concurrent runtime system in it. However, the choice of using F# does not come without its share of problems, specifically when comparing it to Scala. Unlike Scala, F# uses reified generics which means that generic types are not erased at runtime. This may potentially cause issues in regards to ensuring type-safety and their flexibility. In addition, F# does not support Generalized Algebraic Data Types due to type constraints of the .NET type system which may be an issue when it comes to expressing advanced type behavior.

This thesis starts out with an explanation of the theoretical background that is required for understanding such a runtime system. The described problems will be solved by using advanced design patterns and emulating type erasure at runtime. The implementation of the system will be shown realizing the design requirements and constraints and then it will be evaluated by benchmarking aspects of concurrent programming. At last, the project will be concluded upon together with the results.

1.1 Objectives

The following 3 objectives is a formalization of the goals described in the introduction. The objectives will be reprised throughout the thesis when relevant and to which extent they have been satisfied will be presented in Chapter 8.

O1: Guaranteeing type-safety
FIO should guarantee type-safety such that it is not possible to compile a program with type errors present.

O2: Developer-friendly API
FIO's API should be intuitive, small, composable and make it simple to create highly concurrent programs.

O3: Better scalability compared to OS threads
FIO should provide better scalability through its own green thread implementation when compared to OS threads.

1.2 Outline

This thesis consists of 8 chapters including the introduction. The following is a brief description of each chapter.

Chapter 2: A Quick Tour of FIO
Chapter 2 aims to give a practical introduction to FIO, primarily showcasing its strengths and how it can be used in application development. 7 programming examples are presented.

Chapter 3: Background
Chapter 3 provides the necessary theoretical background knowledge that is required to understand the contents of the thesis.

Chapter 4: Design
Chapter 4 describes the design of FIO by considering several design options that aim to maximize the objectives. This is achieved with the knowledge gained from Chapter 3.

Chapter 5: Implementation
Chapter 5 describes technical implementation details of FIO by realizing the design decisions made in Chapter 4.

Chapter 6: Evaluation
Chapter 6 describes how FIO is evaluated alongside an examination of the final results.

Chapter 7: Future Work
Chapter 7 describes improvements and further enhancements to FIO that could be realized in the future.

Chapter 8: Conclusion
Chapter 8 concludes the thesis by acknowledging how well the results from Chapter 7 alongside the design and implementation support the thesis objectives.

Chapter 2

A Quick Tour of FIO

The purpose of this chapter is to provide a practical introduction of FIO's capabilities before technical details are discussed. It should be emphasized that it is not the intention for the reader to grasp everything in this section, but rather get an overall idea of the abilities of the library. 7 programming examples will be presented with each illustrating a particular use case. First, simple succeeding and failing programs will be presented, followed by a program that illustrates basic concurrency. Next, common scenarios in application development is presented, namely how to handle functions that might fail and how to choose between two functions based on the one that finish executing first. Then a message passing program will be shown using communication channels, and at last a program demonstrating the scalability and high concurrency level of FIO is presented.

The most simple computation that can be achieved using FIO is succeeding or failing with a value. A program that succeeds with the value `"Hello world!"` is presented in Listing 1.

On line 1 a FIO effect called `hello` is declared. This is a functional effect which describes a program that succeeds with `"Hello world!"`. A functional effect has to be interpreted to get its result. On line 2 FIO's advanced runtime is used to interpret the `hello` effect and a handle to a fiber is returned called `fiber`. A fiber is a green thread and multiple fibers can be spawned within a system thread. More precisely, a green thread is a thread that is scheduled by a runtime rather than the OS, in this case FIO's advanced runtime. The `fiber` handle is a reference to the green thread that is interpreting `hello`. The `fiber` handle is used to await the result of the interpretation on line 3, and on line 4 `Ok "Hello world!"` is printed to the console. To fail with a value, `succeed` can be replaced with `fail` on line 1 and the printed result would then be `Error "Hello world!"` instead.

```
1  let hello : FIO<string, obj> = succeed "Hello world!"
2  let fiber : Fiber<string, obj> = Advanced.Runtime().Run hello
3  let result : Result<string, obj> = fiber.Await()
4  printfn $"%A{result}"
```

Listing 1: A type annotated F# program that succeeds with the value `"Hello world!"`

One may notice that a lot of type information is present in the example shown in Listing 1. In fact, this is not required due to the strong type inference of the F# compiler and can therefore be removed without losing any type-safety. This makes FIO possess strong type-safety, making it impossible to compile a program with type errors. A type inferred version of Listing 1 can be found in Listing 2. These programs are identical and therefore returns the same result.

```
1  let hello = succeed "Hello world!"
2  let fiber = Advanced.Runtime().Run hello
3  let result = fiber.Await()
4  printfn $"%A{result}"
```

Listing 2: A type inferred F# program that succeeds with the value `"Hello world!"`

FIO makes it straightforward to not only create type-safe programs, but type-safe and concurrent programs at once. For example, to spawn and await a fiber that concurrently succeeds with some value, the few lines of code shown in Listing 3 is all that is needed. In the case of this particular program, the fiber succeeds with the value 42 and prints `Ok 42` as the result. The result of an interpretation – succeeding or failing – is not limited to primitive types, but can be used with any type, let it be tuples, collections or custom types.

```
1  let spawner = spawn (succeed 42) >> fun fiber ->
2                await fiber >> fun result ->
3                succeed result
4  let fiber = Advanced.Runtime().Run spawner
5  let result = fiber.Await()
6  printfn $"%A{result}"
```

Listing 3: A program spawning a concurrent fiber that is awaited for its success result of 42

FIO is easily applicable in a wide range of software engineering scenarios. For example, consider the two functions `readFromDatabase` and `awaitWebservice` that retrieves data from a database and awaits data from a web-service respectively. Their implementation is presented in Listing 4. The functions do not connect to any services directly but instead tosses a coin on whether they should succeed or fail. Type annotations are visible to make it easier to see what is going on with the types.

Consider the situation where the functions are integrated into an existing application that uses the `Error` type for throwing errors. As the two functions may fail with different types, they have to be aligned with the rest of the application. In such a scenario, the `attempt` function can be used to catch errors and align them as seen on line 20. `attempt` tries to execute `readFromDatabase`, however if an error is thrown, it is passed along to the continuation function as `err`. The continuation function has the task of aligning the error by passing it further to the `Error` type, in this case `DbError`. An identical situation takes place on line 23.

The program executes `databaseResult` and `webserviceResult` sequentially and uses the `zip` function to return a tuple of results, however if any error is thrown, `attempt` is used to succeed with some default data as seen on line 27. For example, if both of the services succeed `Ok ("data", 'S')` will be returned. If either of the services fail, `Ok ("default", 'D')` is returned instead. This way, it is impossible for the program to fail. This shows how FIO enforces type-safety and how effects with different types can be combined.

```
let readFromDatabase : FIO<string, bool> =
  let rand = Random()
  if rand.Next(0, 2) = 0 then
    succeed "data"
  else
    fail false

let awaitWebservice : FIO<char, int> =
  let rand = Random()
  if rand.Next(0, 2) = 1 then
    succeed 'S'
  else
    fail 404

type Error =
  | DbError of bool
  | WsError of int

let databaseResult : FIO<string, Error> =
  attempt readFromDatabase (fun err -> fail (DbError err))

let webserviceResult : FIO<char, Error> =
  attempt awaitWebservice (fun err -> fail (WsError err))

let program : FIO<(string * char), Error> =
  let result = zip databaseResult webserviceResult
  attempt result (fun _ -> succeed ("default", 'D'))

let fiber = Advanced.Runtime().Run program
let result = fiber.Await()
printfn $"%A{result}"
```

Listing 4: A program simulating the retrieval of data from a database and web-service with error handling

As another scenario, consider the need to request some data from a server, where two servers residing in different regions of A and B are available. Depending on the users physical distance to each server, one or the other might be faster, however the fastest one is always desired for performance. Such a scenario is shown in Listing 5. The functions of `serverRegionA` and `serverRegionB` will be delayed to simulate one server being faster than the other.

The program simply uses the `race` function to race the two effects and the result of the effect that finishes executing first is returned as the result. The type of the effects passed to `race` are required to be identical, as either of their results can

be returned. The type of serverRegionA and serverRegionB is FIO<string, obj> as they both succeed with a string and can not fail. Naturally, the type of program is thus identical as well.

```
let serverRegionA =
  let rand = Random()
  fio (fun _ ->
  succeed (Thread.Sleep(rand.Next(0, 101))))
  >> fun _ ->
  succeed "server data (Region A)"

let serverRegionB =
  let rand = Random()
  fio (fun _ ->
  succeed (Thread.Sleep(rand.Next(0, 101))))
  >> fun _ ->
  succeed "server data (Region B)"

let program = race serverRegionA serverRegionB

let fiber = Advanced.Runtime().Run program
let result = fiber.Await()
printfn $"%A{result}"
```

Listing 5: A program simulating a race between two servers – the result of the fastest one is used

When using FIO, two concurrently running fibers can communicate through message passing. This is accomplished through a channel where messages can be sent and received. For example, a simple pingpong program consisting of a *pinger* and *ponger* effect is presented in Listing 6.

pinger sends a ping message to *ponger* by sending "ping" through chan1. *ponger* is awaiting retrieval of a message on chan1 concurrently, and once received sends "pong" as a reply to *pinger* through chan2. Once *pinger* receives the pong message on chan2 the program terminates.

The result of each effect is Ok () as stop is an alias for succeeding with the Unit type. ||| interprets the two effects in parallel so the printed result of this program would be Ok ((), ()). Returning () is useful in situations where the result is not of importance but rather the computations taking place, which in this case would be the message passing.

```
let pinger chan1 chan2 =
  let ping = "ping"
  send ping chan1 >> fun _ ->
  printfn $"pinger sent: %s{ping}"
  receive chan2 >> fun pong ->
  printfn $"pinger received: %s{pong}"
  stop

let ponger chan1 chan2 =
  receive chan1 >> fun ping ->
  printfn $"ponger received: %s{ping}"
  let pong = "pong"
  send pong chan2 >> fun _ ->
  printfn $"ponger sent: %s{pong}"
  stop

let chan1 = Channel<string>()
let chan2 = Channel<string>()
let pingpong = pinger chan1 chan2 ||| ponger chan1 chan2

let fiber = Advanced.Runtime().Run pingpong
let result = fiber.Await()
printfn $"%A{result}"
```

Listing 6: Two fibers running concurrently passing "ping" and "pong" messages between each other

However, where FIO really shines is with its scalability of being able to spawn thousands of fibers concurrently. For example, consider the program in Listing 7. This program spawns 100.000 concurrent fibers that each send one message to a single receiving fiber. Once the fiber has received all 100.000 messages, the program is terminated. The program may not seem useful in practice, but it demonstrates FIO's ability to spawn and schedule 100.000 green threads, using a simple and concise syntax, that can utilize massive concurrency in barely 29 lines of code.

It should also be mentioned, that there is no upper bound on how many fibers can be spawned at once. The optimal amount of fibers depends on the hardware that FIO is being used on and certain characteristics of the program.

```
let sender chan id =
  let msg = 42
  send msg chan >> fun _ ->
  printfn $"Sender[%i{id}] sent: %i{msg}"
  stop

let rec receiver chan count =
  if count = 0 then
    stop
  else
    receive chan >> fun msg ->
    printfn $"Receiver received: %i{msg}"
    receiver chan (count - 1)

let rec create chan count acc =
  if count = 0 then
    acc
  else
    let newAcc = sender chan count |||* acc
    create chan (count - 1) newAcc

let fiberCount = 100000
let chan = Channel<int>()
let acc = sender chan fiberCount |||* receiver chan fiberCount
let program = create chan (fiberCount - 1) acc

let fiber = Advanced.Runtime().Run program
let result = fiber.Await()
printfn $"%A{result}"
```

Listing 7: A program that spawns 100.000 concurrent fibers all sending messages through the same channel

To summarize, how to succeed with values have been shown in Listing 1 with type annotations and Listing 2 with inferred types. An explanation of how to fail with a value was given as well. A simple example of how to spawn a fiber and await its result was shown in Listing 3. Next, two examples of common software engineering scenarios was shown, namely how to handle the potential failure of functions in Listing 4 and how to race two functions as shown in Listing 5. Then how two concurrent fibers can communicate with the use of message passing through channels was shown in Listing 6, and finally an example of a highly concurrent program spawning 100.000 fibers was presented in Listing 7. These examples should be able to provide some insight into how FIO works and which sort of functionality it provides. In the upcoming chapter, the background of

the thesis will be presented consisting of the foundational theory behind FIO, as well as some examples of related toolkits such as Cats Effect and ZIO for Scala.

Chapter 3

Background

This chapter aims to provide the necessary knowledge that is required to understand the contents of this thesis. Firstly, the notion of functional programming and related theoretical concepts will be explained, including pure functional programming, functional effects and the IO monad. Afterwards, Domain-Specific Languages will be introduced and compared to that of General-Purpose Languages. In addition, the idea of a Domain-Specific Language that implements an interpreted virtual machine will be introduced as well. At last, the Cats Effect and ZIO toolkits for creating concurrent applications will be introduced together with the existing yet experimental library FIO (DC).

3.1 Functional programming theory

Functional programming (FP) is a declarative programming paradigm where programs are created by applying and composing mathematical functions. This is in contrast to imperative programming where programs are written in terms of steps that the computer must execute to accomplish some goal. Fundamental ideas of functional programming include functions being treated as variables (first-class functions) and function parameters can be functions themselves (higher-order functions).

3.1.1 Pure functional programming

Pure functional programming deals with total functions that have no side effects – also called pure functions. A side effect is present when a function communicates with an external environment. This could be printing to the console, writing or reading from a database, or using a mutable variable declared outside the scope of the function. A pure function depends entirely on its input and will always produce the same output based on the input. This is not the case in impure functional programming as functions may be partial or include side effects.

An example of a pure function in F# is shown in Listing 8. This function is pure as it adheres to the rules above.

```
1  let pure (x : int) (y : int) : int =
2    x + y
```

Listing 8: A pure function in F#

In contrary, the function shown in Listing 9 is impure. This is because the function is dependent on the mutable variable y which could change at any point in time.

```
1  let mutable y : int = 42
2  let impure (x : int) : int =
3    x + y
```

Listing 9: An impure function in F#

This leads to a key term – referential transparency. An expression is said to be referentially transparent if it can be replaced with its resulting value without changing the behavior of the program. If a function is referentially transparent, then it is also pure. Referential transparency provides advantages such as making it possible to execute functions in any order and get the same result, as well as being able to safely substitute pure functions. If an expression is not referentially transparent, then it is referentially opaque.

3.1.2 Functional effects

In pure functional programming, no side effects are allowed as they make it difficult to reason about correctness and makes referential transparency impossible. However, programming without side effects is a major limitation. How would one read or write to a database, if side effects are not allowed?

Two worlds exist in functional programming – the pure and the impure (or external) world. In the pure world, all computations are pure and there exists no side effects. Conversely, in the impure world, side effects are allowed.

A functional effect is an immutable value that models impure side effects in the pure world – it turns impure side effects into pure side effects. When a functional effect is created, it is not executed, rather, it is only allocated in memory. As a consequence, functional effects have to be executed – they are lazily evaluated. Functional effects can be viewed as descriptions of programs with side effects that have yet to be executed, but can be executed given some runtime. This makes an impure expression referentially transparent, and therefore also pure, meaning that it is with functional effects possible to have side effects in a purely functional language.

Recall that in Chapter 2, multiple functional effects were showcased being executed by FIO's advanced runtime.

3.1.3 The IO monad

The IO monad is a construct that transforms impure expressions with side effects into pure expressions with side effects. It achieves this by wrapping the impure expression into an `IO` type which is treated as a functional effect. The IO in its name stands for "Input/Output" as common side effects include read and write operations with external environments.

Haskell is an example of a purely functional language where the IO monad is present in the standard library. This is not the case for Scala and thus this language needs an external implementation such as Cats Effect or ZIO.

To provide further intuition of the IO monad, a simple demonstration of the Cats Effect IO monad follow. The example shows how the monad can turn referentially opaque expressions referentially transparent. Consider the Scala program shown in Listing 10 which prints `"Hello world!"` to the console once.

```
1 val x = println("Hello world!")
2 (x, x)
```

Listing 10: A Scala program that prints `"Hello world!"` once

If the expression `x` is referentially transparent, then the behavior of the program will not change if `x` is replaced by its resulting value `println("Hello world!")`. Consider the same Scala program, but now with the expression replaced by its value as presented in Listing 11.

```
1 (println("Hello world!"), println("Hello world!"))
```

Listing 11: A Scala program that prints `"Hello world!"` twice

When this program is run `"Hello world!"` is printed twice compared to once for the previous program. Hence, this proves that the expression `val x = println("Hello world!")` is referentially opaque.

The IO monad provided by Cats Effect can be used to wrap the expression and turn it into a functional effect, thereby making it referentially transparent. Consider the modified version of Listing 10 shown in Listing 12.

```
1 import cats.effect.IO
2
3 val x = IO(println("Hello world!"))
4 (x, x)
```

Listing 12: A Scala program that prints nothing

When this program is run, `"Hello world!"` will not be printed to the console as in the previous version, hence this program has no side effects. The program

returns `(cats.effect.IO[Unit], cats.effect.IO[Unit])`, a tuple of the IO type, or in other words – a tuple of functional effects.

Similarly, a new version of Listing 11 is shown in Listing 13. When run, this program also produces `(cats.effect.IO[Unit], cats.effect.IO[Unit])` as a final result with no printing to the console, and thus the expression `x` is now referentially transparent thanks to the IO monad.

```
1  import cats.effect.IO
2
3  (IO(println("Hello world!")), IO(println("Hello world!")))
```

Listing 13: Another Scala program that prints nothing

In a similar fashion to the `IO` type, all effects created in FIO are of the type `FIO` as previously shown in Listing 1. More precisely, it can be seen on line 1 that the functional effect `hello` has the type `FIO<string, obj>`.

3.2 Domain-specific languages

When the IO monad is used, it is used as a term of the language that wraps around some expression, which is then given to some interpreter that executes and performs the actual effects. For this reason, one usually ends up developing some kind of API that provides a variety of useful effects. In the context of monadic programming, such an API is usually implemented in terms of an internal (also called embedded) Domain-Specific Language.

A Domain-Specific Language (DSL) is a computer language that is specialized for problem solving in a particular application domain. A DSL is the opposite of a General-Purpose Language (GPL) which is applicable across a broad variety of application domains. This includes General-Purpose programming languages such as Java and Haskell. It should be noted that the term "computer language" is not restricted to programming languages as GPLs also include markup and modeling languages such as XML and UML. Examples of DSLs include HTML for structuring webpages and MATLAB for scientific computing.

The advantage of using DSLs over GPLs is that they are usually more safe and practical to use in their particular application domain. Moreover, DSLs are usually smaller and higher-level languages than GPLs due to only needing

specific features and constructs for their domain. DSLs can further be divided into two groups, internal and external DSLs. [Fow19]

Internal DSL An internal DSL is embedded inside a host language to provide the feeling of a language that is used for a particular application domain. A potential limitation with internal DSLs is that the user is limited by the syntax and programming paradigm of the host language. On the other hand, the user is available to use features of the host language which might be beneficial. [Fow06]

External DSL An external DSL has its own syntax and requires a parser to process it. This gives the language developer more refined control over the DSL, however whether it is worth to spend the time creating a parser depends on the particular use case and domain of the language.

Some examples from an API developed in terms of an internal DSL has already been shown in Chapter 2. For example, in Listing 3 the `spawn`, `await` and `succeed` functions are all part of FIO's effect API.

3.2.1 Domain-specific language for an interpreted virtual machine

A virtual machine (VM) is a software-defined machine that emulates functionality of a computer system. In other words, the hardware components of the machine are defined using software. For instance, components such as the CPU and its Instruction Set Architecture (ISA) are defined using computer code. A popular and widely used VM is the Java Virtual Machine (JVM) used for programming languages such as Scala and Java.

Languages that run on the JVM compile down to instructions (or opcodes) that are collectively known as Java bytecode. These opcodes are then executed by the JVM to perform computations. Example of opcodes supported by the JVM are presented in Table 3.1.

Opcode	Summary
aconst_null	Push null on the stack
aload	Load reference from a local variable
astore	Store reference into a local variable
athrow	Throw an excepton or error

Table 3.1: Opcode examples supported by the JVM

The idea of having a virtual machine execute opcodes to perform computations can be combined with having an internal DSL describe an API of effects. More precisely, just like how constructs in Java and Scala compile down to Java bytecode, effects from the API can compile down to opcodes which are then interpreted by a virtual machine. In fact, for FIO, this is precisely the case. The previously mentioned effects of `spawn`, `await` and `succeed` are all built by composing one or more internal opcodes, which are then given to FIO's runtime (the virtual machine) for interpretation.

3.3 Functional programming toolkits for concurrent applications

Cats Effect and ZIO are toolkits for Scala that provide an implementation of the IO monad alongside types for asynchronous and concurrent programming. FIO (DC) is an implementation of the IO monad – similar to that of ZIO – but for F# rather than Scala.

3.3.1 Cats Effect

Cats Effect is a high-performance, asynchronous library for building applications in a purely functional style. It provides an implementation of the IO monad for capturing and controlling side effects within a resource-safe, typed context with support for concurrency and coordination. Libraries like Cats Effect are also referred to as "effect systems". Cats Effect is widely used by well-known companies such as Comcast, ING Bank, Prezi and Zendesk. [Typb] [Typc]

Since Cats Effect is a library for asynchronous and concurrent programming, it provides an alternative to the OS thread. Such an alternative is referred to as a "green thread" or "fiber" as seen for FIO as well. The primary difference between JVM threads and green threads is that the latter is not scheduled by the OS but by the library and may thus be more light-weight than JVM threads. [Typa]

An example from the Cats Effect webpage is found in Listing 14. The example wraps the IO monad around a print expression and then saves it in the `program` effect. Next, the effect is executed by the `unsafeRunSync()` function and `"Hello world!"` is printed once to the console.

```scala
import cats.effect.unsafe.implicits._
import cats.effect.IO

val program = IO.println("Hello world!")
program.unsafeRunSync()
```

Listing 14: Example of creating an effect and executing it using Cats Effect

3.3.2 ZIO

ZIO is a framework for asynchronous and concurrent programming that is based on pure functional programming. The core concept of ZIO is `ZIO`, an effect type that is inspired by Haskell's IO monad. Essentially, the idea of ZIO is to make it possible for Scala to be used as a pure functional language. Similar to Cats Effect, ZIO is also referred to as an effect system and uses green threads. ZIO is broadly used among companies such as Adidas, eBay, Wolt and Zalando. [Maib] [Mai22]

An example of using ZIO from ZIO's documentation can be seen in Listing 15. In the example, an effect called `myAppLogic` is created that asks for the users name and simply prints it to the console. As the class inherits from `zio.App`, the effect is automatically executed by the ZIO runtime. [Maia]

```scala
import zio._
import zio.console._

object MyApp extends zio.App {

  def run(args: List[String]) =
    myAppLogic.exitCode

  val myAppLogic =
    for {
      _    <- putStrLn("Hello! What is your name?")
      name <- getStrLn
      _    <- putStrLn(s"Hello, ${name}, welcome to ZIO!")
    } yield ()
}
```

Listing 15: Example of creating an effect and executing it using ZIO

3.3.3 FIO (DC)

FIO (DC) is a framework created by Daniel Chambers, with the intent of creating the ZIO framework for F# . According to the README of the source repository, the current state of the project is experimental and the latest update to the framework is from 2019. An immediate difference when compared to the other libraries is that FIO (DC) does not currently support any type of green thread, it solely relies on OS threads for concurrency. It does not support message passing either.

The current version supports some similar functionality to ZIO, and it also supports the usage of F# computation expressions for a neater syntax when combining effects. An example similar to the one for ZIO in Listing 15 can be seen in Listing 16. [Cha19]

```
1  open FSharp.Control.FIO
2  
3  [<EntryPoint>]
4  let main _ =
5    let effect = fio {
6      do! Console.writeLine "Hello! What is your name?"
7      let! name = Console.readLine()
8      do! Console.writeLine $"Hello, %s{name}, welcome to FIO!"
9    }
10   FIO.runFIOSynchronously (RealEnv()) effect |> ignore
11   0
```

Listing 16: Example of creating an effect and executing it using FIO (DC)

3.4 Summary

In this chapter, key concepts from the theory behind functional programming was presented, such as what functional programming is and related fundamental concepts. Additionally, the differences between impure and pure functional programming was explained together with side effects and referential transparency. Then functional effects were introduced with how they transform impure expressions into pure expressions. Next, the concept of the IO monad was introduced as well and how it relates to functional effects together with a simple demonstration.

Domain-Specific Language was presented and explained how it is conventionally used together with the IO monad in regards to building an API. The usage of an internal DSL together with a virtual machine for interpretation of opcodes were presented and had their similarities aligned with the IO monad as well.

Finally, two popular library implementations of the IO monad with the names of Cats Effect and ZIO for Scala were introduced together with green threads. Simple usage examples were presented for each library as well. At the very end, the experimental library of FIO (DC) for F# was introduced and discussed how it relates to ZIO and a simple usage example was presented as well.

In the next chapter, the design decisions and considerations of FIO will be discussed alongside more in-depth technical details.

Chapter 4

Design

In this chapter, the design decisions made throughout the project along with some technical details will be discussed. The chapter divides the project into two sections, effect system and runtime system. In the effect system section, the design will be described for the effect API, effect type, the interpreter structure and finally effect structure with channels, fibers and opcodes. In a similar manner for the runtime system, the naive interpreter, evaluation and blocking workers, intermediate and advanced interpreter, as well as debugging tools for data structure monitoring and dead lock detection will be discussed.

4.1 Effect system

With the knowledge provided by Chapter 3, it is now possible to start reasoning about the design of an effect system that supports the functionality seen in Chapter 2.

4.1.1 Effect API

The effect API should provide simple and concise functions for the user. This is accomplished by designing functions that focus on a single task and accomplishes that task well, also known as the KISS principle. The API will be designed as a set of functions that make up an internal DSL with F# as its host language.

To recall some of the available functions that have already been seen, consider the same program from Chapter 2 (Listing 3) as seen in Listing 17, albeit with runtime elements removed. To elaborate on this program in further detail, this effect spawns a fiber using the `spawn` function from the API. The result of this function is then unwrapped by using the infix sequencing function >> to make the `fiber` handle directly accessible. The fiber is then awaited by using the `await` function on the handle which result is sequenced once again to unwrap it. At last, the `succeed` function is used to return `result` as a success value.

```
1 let program =
2   spawn (succeed 42) >> fun fiber ->
3   await fiber >> fun result ->
4   succeed result
```

Listing 17: A program spawning a concurrent fiber that is awaited for its success result of 42

The way that the API is used has a natural flow to it using the sequencing function to combine multiple effects into a single effect. It is therefore possible to create complex effects in few lines of code as previously proven in Chapter 2. This supports objective *O2*.

Looking at Listing 17, it is clear that a lot of logic is happening behind the scenes in just four lines of code. The reason that this is possible, is that each function in the API is built by composing one or more opcodes or even API functions as mentioned in Chapter 3. In other words, the functions are built

similarly to how combined effects are built, however the difference is that they may be composed of both opcodes and API functions.

This makes a few challenges arise, mainly in regards to the balance of how much logic the opcodes should contain versus the API functions. In addition, the available opcodes may impact the functionality and performance of the API. These problems will be discussed in further detail in Section 4.1.4.3.

In Table 4.1 an overview of the available API functions in FIO can be seen. This includes the name of each function, alongside which arguments they take and return. A brief description of each function is present as well. These functions are inspired by functionality found in Cats Effect and ZIO. They are chosen in particular as they provide a strong base for creating concurrent applications, however it is possible to extend the API with more functions if needed. The functions follow the KISS principle well and are therefore deemed to satisfy objective *O2* as well. [Maib] [Typd]

FIO (DC) contains some of the same API functions as FIO, however not including equivalent functions to `send`, `receive`, `spawn`, `await` and `race`. This is due to FIO (DC) not using similar constructs to channels or fibers.

Function	**Arguments & Return**	**Description**
`fio`	`func: (Unit -> 'R)` Returns: `FIO<'R, 'E>`	Transforms the expression `func` into a functional effect `FIO`.
`succeed`	`result: 'R` Returns: `FIO<'R, 'E>`	Succeeds with the result `result`.
`fail`	`error: 'E` Returns: `FIO<'R, 'E>`	Fails with the error `error`.
`stop`	Returns: `FIO<Unit, 'E>`	Succeeds with the `Unit` type.

`send`	`value: 'R`
	`chan: Channel<'R>`
	Returns: `FIO<'R, 'E>`
		Sends `value` into the channel `chan`.

`receive`	`chan: Channel<'R>` Returns: `FIO<'R, 'E>`	Awaits and returns data received from channel `chan`.

`spawn`	`eff: FIO<'R1, 'E1>`
	Returns:
	`FIO<Fiber<'R1, 'E1>, 'E>`
		Spawns and returns a fiber that interprets `eff` concurrently.

<table>
<tr><td>await</td><td>fiber: Fiber<'R, 'E>
Returns: FIO<'R, 'E></td><td>Awaits and returns the result of fiber.</td></tr>
<tr><td>>></td><td>eff: FIO<'R1, 'E>
cont:
('R1 -> FIO<'R, 'E>)
Returns: FIO<'R, 'E></td><td>Infix function that sequences two effects. First eff is interpreted with its result passed to cont which then is returned. Errors are returned immediately.</td></tr>
<tr><td>|||</td><td>eff1: FIO<'R1, 'E>
eff2: FIO<'R2, 'E>
Returns:
FIO<'R1 * 'R2, 'E></td><td>Infix function that interprets eff1 and eff2 in parallel and returns a tuple of their results.</td></tr>
<tr><td>|||*</td><td>eff1: FIO<'R1, 'E>
eff2: FIO<'R2, 'E>
Returns:
FIO<Unit, 'E></td><td>Infix function that interprets eff1 and eff2 in parallel and discards each result by returning Unit.</td></tr>
<tr><td>attempt</td><td>eff: FIO<'R, 'E1>
cont:
('E1 -> FIO<'R, 'E>)
Returns: FIO<'R, 'E></td><td>Attempts to interpret eff. If an error occurs it is passed to the continuation. Success are returned immediately.</td></tr>
<tr><td>zip</td><td>eff1: FIO<'R1, 'E>
eff2: FIO<'R2, 'E>
Returns:
FIO<'R1 * 'R2, 'E></td><td>Interprets eff1 and eff2 sequentially and returns the result in a tuple. Errors are returned immediately.</td></tr>
<tr><td>race</td><td>eff1: FIO<'R, 'E>
eff2: FIO<'R, 'E>
Returns: FIO<'R, 'E></td><td>Interprets eff1 and eff2 in parallel. The result of the effect that completes first is returned.</td></tr>
</table>

Table 4.1: Overview of the API functions available in FIO with a brief description

4.1.2 Effect type

In Section 3.1.3 it was mentioned that the FIO type is similar to IO of Cats Effect in the way that it can wrap around an expression and turn it into a functional effect. Secondly, in Section 4.1.1 it was shown how the functions of the effect API both accept and return forms of the type FIO<'R, 'E>, that is functional effects.

An effect type is the type that represents a functional effect and it can be defined in a variety of ways. The approaches of Cats Effect and ZIO are shown below as examples. The types will be represented using F# notation to provide a familiar type signature to reason about.

Cats Effect The effect type of Cats Effect is represented as a data type IO<'A>. IO is a type that is generic over the type parameter 'A which is the type of the returned value in a success scenario.

ZIO The effect type of ZIO is the data type ZIO<'R, 'E, 'A>. 'R is the environment type, for example if the effect requires a console instance to run. 'E is the failure type which is returned if the effect fails, and 'A is the success type which is returned if the effect succeeds. FIO (DC) uses an identical effect type to ZIO.

The effect type for FIO is defined as the type FIO<'R, 'E> where 'R is the type of the returned result in a success scenario and 'E the type of the returned result in a failure scenario. For this reason, the succeed and fail functions from the API return values of the types 'R and 'E respectively. In more technical terms, when an instance of FIO<'R, 'E> is interpreted, it will result in the Result<'R, 'E> type. Result<'R, 'E> can either be Ok with a value of 'R or Error with a value of 'E and thus matches the requirement. This particular type is chosen as it is simple and can be extended if needed. An extension to this type could be similar to the environment type of ZIO. With FIO<'R, 'E> the error type is made explicit, which is not the case with Cats Effect. This makes it possible to see which kind of error an effect might fail with which is convenient.

4.1.3 Interpreter structure

Before the design of the effect structure, channels, fibers and opcodes is discussed, it is necessary to discuss the structure of the interpreter. The design of the interpreter will have direct influence on how these elements will be designed.

The initial choice for designing the interpreter structure was Algebraic Data Types (ADTs), however due to F# not supporting Generalized Algebraic Data

Types (GADTs) this approach did not seem possible. An example of implementing two opcodes using ADTs is shown in Listing 18. The `Success` opcode represents succeeding with a value of type `'R` and `SequenceSuccess` the sequencing of opcodes. The `success` function from the effect API can be seen as an alias of `Success` and the infix operator >> an alias of `SequenceSuccess`. This code does not compile as the type parameter `'R1` is not defined and can not be defined without adding it to the effect type `FIO<'R, 'E>`. Doing this would not be ideal as other opcodes may not need `'R1`.

```
1  type FIO<'R, 'E> =
2      | Success of res: 'R
3      | SequenceSuccess of eff: FIO<'R1, 'E> * cont: ('R1 ->
   ↪  FIO<'R, 'E>)
```

Listing 18: Example of using ADTs to implement two opcodes `Success` and `SequenceSuccess` (This code does not compile)

As a secondary approach, using an inheritance hierarchy of the opcodes was attempted. Unfortunately, it was found that pattern matching against a type with generic parameters did not work as expected. In Listing 19 an example implementation can be seen. On line 12 the pattern matching on `eff` raises a compiler warning saying that the pattern matching is incomplete. This is because the type parameters of the opcodes needs to be known at compile time which makes this approach impractical as well.

```
1  type [<AbstractClass>] FIO<'R, 'E>() = class end
2  and Success<'R, 'E>(res : 'R) =
3    inherit FIO<'R, 'E>()
4    member _.Res = res
5  and SequenceSuccess<'R1, 'R, 'E>(eff : FIO<'R1, 'E>,
6      cont : 'R1 -> FIO<'R, 'E>) =
7    inherit FIO<'R, 'E>()
8    member _.Eff = eff
9    member _.Cont = cont
10
11 let rec interpreter (eff : FIO<'R, 'E>) : Result<'R, 'E> =
12   match eff with
13   | :? Success<'R, 'E> as success -> Ok success.Res
14   | :? SequenceSuccess<'R1, 'R, 'E> as seque -> // logic <snip>
```

Listing 19: Example of an inheritance hierarchy implementing two opcodes `Success` and `SequenceSuccess` (This code does not work)

Due to the above issues, a post on StackOverflow was created with the hopes of gaining some insight on how the wanted structure could be designed. The post was answered by Tomas Petricek, a lecturer at the University of Kent and partner at fsharpWorks. Tomas suggested to use the object-oriented visitor pattern as that would seemingly be able to express the desired structure. [Lar22] [Pet]

In addition, an alternative approach using ADTs was suggested by Alceste. The idea with this approach is to replace `'R1` with the `obj` type and then use type casting to downcast `obj` to its specific type at runtime. This can be seen as a way to mimic type parameter erasure as F# has reified generics. Reifeid generics means that generic types are not erased after compilation which makes it difficult to have runtime type conversions. This has not been an issue for FIO (DC) as this library does not have a notion similar to opcodes, but rather use API functions directly. This means that the type parameters can freely be defined in the function signatures rather than being limited to a base type. Also, some interpreter implementations of FIO will split effects into multiple effects, making it required that they have the same type. FIO (DC) does not use this strategy so having the same type for each effect is not a strict necessity. In the end, the visitor pattern and algebraic data types with type casting were chosen as the two considered designs.

4.1.3.1 Visitor pattern

The visitor pattern is an object-oriented, behavioral design pattern that separates the logic from the objects on which they operate. The idea is for each object to accept a *Visitor* class that encapsulates the logic for each object. When the functionality of a specific object is invoked by passing the *Visitor*, the *Visitor* will "visit" the specific type of object and invoke its functionality, thus giving separation between the logic and object.

Using this design pattern the opcodes will be built as an inheritance hierarchy with the effect type as the base class. This gives flexibility to freely express the generic type parameters of the opcodes and not limiting them to the types of the base type. In addition, the *Visitor* will be implemented as an interface that can be passed into an `accept` function of each opcode, where the *Visitor* can invoke its logic for the particular opcode. One problem is, however, that the actual implementation of the visitor pattern can be verbose and require a lot of boilerplate code which will decrease its readability and maintainability.

A complete example of using the visitor pattern with the two opcodes `Success` and `SequenceSuccess` is presented in Listing 20. At line 14, notice that it

is possible for the `SequenceSuccess` opcode to take 3 type parameters rather than 2 like its base type. Additionally, at line 23, a new `Visitor` instance is instantiated. As the nature of the interpreter will be recursive, instantiating a new instance at every recursion may have an impact on performance.

```
type Visitor =
  abstract VisitSuccess<'R, 'E> : Success<'R, 'E> ->
    Result<'R, 'E>
  abstract VisitSequenceSuccess<'R1, 'R, 'E> :
    SequenceSuccess<'R1, 'R, 'E> -> Result<'R, 'E>

and [<AbstractClass>] FIO<'R, 'E>() =
  abstract Accept<'R, 'E> : Visitor -> Result<'R, 'E>

and Success<'R, 'E>(res : 'R) =
  inherit FIO<'R, 'E>()
  member _.Res = res
  override this.Accept visitor =
    visitor.VisitSuccess this

and SequenceSuccess<'R1, 'R, 'E>(eff : FIO<'R1, 'E>,
    cont : 'R1 -> FIO<'R, 'E>) =
  inherit FIO<'R, 'E>()
  member _.Eff = eff
  member _.Cont = cont
  override this.Accept visitor =
    visitor.VisitSequenceSuccess this

let visitor = {
  new Visitor with
    member _.VisitSuccess (success : Success<'R, 'E>) :
      Result<'R, 'E> =
      Ok success.Res
    member _.VisitSequenceSuccess (sequence :
      SequenceSuccess<'R1, 'R, 'E>) : Result<'R, 'E> =
      // logic <snip>
  }
```

Listing 20: Example of using the visitor design pattern with two opcodes `Success` and `SequenceSuccess`

4.1.3.2 Algebraic data types with type casting

An Algebraic Data Type (ADT) is a structured type that is made by composing other types. Two kinds of ADTs exist – product types such as tuples and sum types like discriminated unions of which is the available implementation in F#.

As shown previously, the approach of using discriminated unions in Listing 18 was not feasible. The idea with this new approach is to replace 'R1 with the obj type – the base type of all types. As obj is not a generic type parameter, no changes are required to FIO<'R, 'E> and thus the code is now able to compile. The obj type can then be downcasted to its exact type at runtime.

To imitate generic type erasure, it is necessary to use FIO<obj, obj> rather than FIO<'R, 'E> internally. This requires for FIO<'R, 'E> to be upcasted to FIO<obj, obj> at the beginning of interpretation. FIO<obj, obj> would then be downcasted to its specific types when the interpreted result is returned to the user. This is viable because the specific types are not required during interpretation of an opcode and is only relevant when reaching the user. This gives the advantage of all opcodes having equivalent types internally, which will make it more simple to develop advanced interpretation techniques.

An example of this is presented in Listing 21. Notice how the interpreter function uses the UpcastResult and UpcastError functions on the effect before it is passed to lowLevelInterpreter at line 25. interpreter is the function available to the user while lowLevelInterpreter is only used internally.

Using this approach will provide a more concise implementation that is easier to understand compared to the visitor pattern while also making it easier to implement advanced interpreters. However, a disadvantage is that the excessive amount of type casting will with high certainty have a measurable impact on performance.

Ultimately, despite the performance concerns, the approach of using ADTs with type casting will be used instead of the visitor pattern due to the more concise implementation and ease of development for advanced interpretation.

```
type FIO<'R, 'E> =
  | Success of res: 'R
  | SequenceSuccess of eff: FIO<obj, 'E> * cont: (obj -> FIO<'R,
  ↪  'E>)

  member this.UpcastResult<'R, 'E>() : FIO<obj, 'E> =
    match this with
    | Success res -> Success (res :> obj)
    | SequenceSuccess (eff, cont) ->
        SequenceSuccess (eff, fun res ->
        (cont res).UpcastResult())

  member this.UpcastError<'R, 'E>() : FIO<'R, obj> =
    match this with
    | Success res -> Success res
    | SequenceSuccess (eff, cont) ->
        SequenceSuccess (eff.UpcastError(), fun res ->
        (cont res).UpcastError())

let rec internal lowLevelInterpreter (eff : FIO<obj, obj>) :
↪  Result<obj, obj> =
  match eff with
  | Success res -> Ok res
  | SequenceSuccess (eff, cont) -> // logic <snip>

let interpreter (eff : FIO<'R, 'E>) : Result<'R, 'E> =
  let result = lowLevelInterpreter
  ↪  (eff.UpcastResult().UpcastError())
  match result with
  | Ok res -> Ok (res :?> 'R)
  | Error err -> Error (err :?> 'E)
```

Listing 21: Example of using algebraic data types with type casting to implement two opcodes `Success` and `SequenceSuccess`

4.1.4 Effect structure

Now that the interpreter structure has been determined, the structural components that make up effects can be discussed and designed. This includes a formal introduction to the design of channels, fibers and opcodes.

4.1.4.1 Channels

A channel is a communication medium used for message passing. In more precise terms, a channel is a type `Channel<'R>` that handles messages of type `'R` sorted after the FIFO principle (a queue). A channel provides two functions for the user. `Add` is a non-blocking function that adds a message to the channel, which then can be received by using the blocking `Take` function. The idea is for the `send` and `receive` API functions to wrap around these two functions to provide functional effects that use channels.

As an opcode requires its arguments to be upcasted due to the chosen interpreter structure, a channel must be able to be upcasted as well. This is done with the internal `Upcast` function which returns a new channel of the type `Channel<obj>` however with reference to the same data as the original channel. An example of a simple channel implementation can be seen in Listing 22.

```
1  type Channel<'R> (bc : BlockingCollection<obj>) =
2    new() = Channel(new BlockingCollection<obj>())
3    member internal _.Upcast() = Channel<obj>(bc)
4    member _.Add(value : 'R) = bc.Add value
5    member _.Take() : 'R = bc.Take() :?> 'R
```

Listing 22: An example of a Channel implementation that handles messages of type `'R`

4.1.4.2 Fibers

A fiber is a green thread that can be seen as a schedulable computation, however instead of being scheduled by the OS it is scheduled by a runtime. In the case of FIO, the way that a fiber is scheduled depends on which runtime is being used. This will be explained in further detail in Section 4.2.

In FIO, a fiber represents the concurrent interpretation of an effect. Once the effect is interpreted, the fiber is completed with the result of the effect and the result is now available. To retrieve the result, the handle to the fiber that was returned by the interpreter can be awaited, blocking the caller until the result is available. 3 important rules of fibers follow.

R1 A fiber may only be completed once.

R2 If a fiber is being completed concurrently, only one of the completions may be applied.

R3 If a completed fiber is awaited, it will always return the same result.

Implementation details on how these rules are realized will be shown in Chapter 5. Two types of fibers will be designed, an internal LowLevelFiber and a user facing handle Fiber. Use of the Fiber handle has been seen previously in Chapter 2.

LowLevelFiber The internal fiber is designed as the type LowLevelFiber. It takes no type parameters as it works with results of type obj. A LowLevelFiber will never be instantiated directly, instead it will always be converted from an existing Fiber.

Fiber The fiber handle is designed as the type Fiber<'R, 'E>. The only function available to the user is the Await for awaiting on the fibers result of of Result<'R, 'E>. A second internal function exists, ToLowLevel. This function converts the Fiber into a LowLevelFiber with reference to the same data. The Fiber type will solely be used as a handle for the user to await interpretation results.

In practice, two fiber types are not required. Fiber<'R, 'E> could be upcasted to Fiber<obj, obj> instead of having LowLevelFiber as a separate type. However, having separate types were chosen to keep separation between the types that the user can access and what is used internally.

4.1.4.3 Opcodes

Opcodes represent internal, low-level and simple effects that, as explained in Section 4.1.1, will be used as building blocks for the effect API. Even though the opcodes should consist of simple and low-level functionality, it is possible to create complex opcodes. It is a balance between how much logic is kept internally in the opcodes and how much logic is kept in the effect API. There are 3 design possibilities – keeping a minimal set of opcodes, keeping a maximal set of opcodes or an arbitrary set of opcodes.

Minimal set of opcodes This approach keeps the set of opcodes concise and clean and will make the interpreter simple as it will have to support few opcodes. However, the functions in the API may grow large and complex and therefore have decreased performance in comparison to being implemented as opcodes.

Maximal set of opcodes If every effect of the system is implemented as an opcode, it makes it possible to optimize each opcode in the interpreter rather than having to build API functions. This is good for performance, but bad when it comes to maintainability as some opcodes may include large amounts of logic and duplicate logic as well.

Arbitrary set of opcodes The idea is to keep a minimal set of performance critical opcodes that are often used. The API functions will then implement higher-level effects that are less commonly used. This approach attempts to provide balance between the two prior approaches in terms of performance and system complexity.

The third and last approach will be used for designing the opcodes and API functions as this gives the best balance between performance, maintainability and readability.

A brief overview of the designed opcodes can be viewed in Table 4.2. Most of the opcodes have fairly simple effects, such as `NonBlocking` for representing any non-blocking operation and `Success` or `Failure` for representing succeeding and failing with a value respectively. However, there are 3 opcodes that are slightly more complex than the others and may therefore require further explanation, namely `Concurrent`, `SequenceSuccess` and `SequenceError.`

`Concurrent` Together with `AwaitFiber`, `Concurrent` is the core opcode for concurrency in FIO. It firstly interprets the effect `FIO<obj, obj>` concurrently inside the `LowLevelFiber`. The `LowLevelFiber` is converted from `Fiber` which is downcasted from the `obj` argument. `AwaitFiber` can then be used with the returned `Fiber` to await the result. In fact, the `spawn` API function is close to being an alias for `Concurrent`. Implementation details of opcodes will be shown in Chapter 5.

`SequenceSuccess` This opcode is used to compose two effects together as seen previously with >>. The result of the first effect is available to the second effect if the first one succeeds. If any of the effects fail, the error is returned immediately. First `FIO<obj, 'E>` is interpreted, then in a success scenario, the `obj` result will be given to `(obj -> FIO<'R, 'E>)` and the function will be applied, returning `FIO<'R, 'E>`.

`SequenceError` This opcode is similar to `SequenceSuccess`, but with reversed logic. Where `SequenceSuccess` composes two effects on the basis of the first effect succeeding, this opcode composes two effects when the first fails. This also means that if a success scenario happens, the success value is returned immediately.

Previously, an example of spawning a fiber that succeeds with the value of 42 was shown in Listing 17 using the effect API. Now that the opcodes are designed,

it can be shown how the same effect would look translated to opcodes. Consider the opcode translation as shown in Listing 23.

As one may notice, the opcodes are not ideal for building complex effects as there is a lot of visible type casting and parentheses. This is the reason why the effect API is used to encapsulate details of the opcodes and provide a set of developer-friendly functions to satisfy *O2*. It should be noted that all API functions begin with lowercase letters, such as `spawn` and opcodes begin with uppercase letters such as `Concurrent`.

```
1  let program =
2    let fiber = new Fiber<int, obj>()
3    SequenceSuccess (
4      Concurrent (Success 42, fiber, fiber.ToLowLevel()),
5      fun innerFiber -> AwaitFiber ((innerFiber :?>
       ↪  Fiber<int, obj>).ToLowLevel()))
```

Listing 23: A program spawning a concurrent fiber that is awaited for its success result 42 using opcodes

As another example, consider the program previously seen in Listing 4 translated to opcodes in Listing 24. Again, it is immediately noticeable that a lot of details are going on in regards to type casting which makes the code more cryptic and unreadable. If larger and more complex programs were to be created directly using opcodes, the complexity would greatly arise compared to using functions from the API and *O2* would then not be satisfied.

```
let readFromDatabase : FIO<string, bool> =
  let rand = Random()
  if rand.Next(0, 2) = 0 then
    Success "data"
  else
    Failure false

let awaitWebservice : FIO<char, int> =
  let rand = Random()
  if rand.Next(0, 2) = 1 then
    Success 'S'
  else
    Failure 404

let databaseResult : FIO<string, Error> =
  SequenceError (readFromDatabase.Upcast(),
                 fun err -> Failure (DbError (err :?> bool)))

let webserviceResult : FIO<char, Error> =
  SequenceError (awaitWebservice.Upcast(),
                 fun err -> Failure (WsError (err :?> int)))

let program : FIO<(string * char), Error> =
  let result =
    SequenceSuccess (databaseResult.UpcastResult(), fun res1 ->
      SequenceSuccess (webserviceResult.UpcastResult(), fun res2
      ↪  -> Success (res1 :?> string, res2 :?> char)))
  SequenceError (result.Upcast(),
                 fun _ -> Success ("default", 'D'))
```

Listing 24: A program simulating the retrieval of data from a database and web-service with error handling using opcodes

Opcode	Arguments	Description
`Success`	'R	A non-blocking operation that returns a success value.
`Failure`	'E	A non-blocking operation that returns a failure value.

NonBlocking	(unit -> Result<'R, 'E>)	A non-blocking operation that applies the function argument and returns a result.
Blocking	Channel<'R>	A blocking operation that awaits on the channel argument and returns the retrieved message.
SendMessage	'R Channel<'R>	A non-blocking operation that sends a message into the channel then returns the sent message.
Concurrent	FIO<obj, obj> obj LowLevelFiber	A non-blocking operation that interprets an effect concurrently. The result can be awaited from from the fibers.
AwaitFiber	LowLevelFiber	A blocking operation that awaits the result of the fiber argument.
SequenceSuccess	FIO<obj, 'E> (obj -> FIO<'R, 'E>)	A non-blocking operation that interprets two effects in sequence if the first succeeds. Errors are returned immediately.
SequenceError	FIO<obj, 'E> (obj -> FIO<'R, 'E>)	Similar to SequenceSuccess but sequences only if the first is an error. Success are returned immediately.

Table 4.2: Overview of the designed opcodes that were deemed to be critical for performance with a brief description

4.2 Runtime system

It is now possible with the designed effect system to consider how the opcodes should be interpreted by the runtime system. 3 interpreters will be designed. This includes the naive interpreter which is meant to be the simplest, most straightforward interpretation design. An intermediate interpreter will attempt to improve upon the naive by introducing the usage of evaluation and blocking

workers, and finally the advanced interpreter will improve upon the intermediate interpreter by optimizing utilization of the workers.

4.2.1 Naive interpreter

The purpose of the naive interpreter is to be the most simple interpreter that can be implemented. The idea for it is to be used as a baseline interpreter, to see if and how it can be improved upon. This interpreter will not be using fibers as green threads, but as OS threads instead. The reason for this is to compare the use of OS threads versus green threads that are scheduled by FIO. The intermediate and advanced interpreter will take advantage of fibers as green threads by letting constructs known as evaluation and blocking workers schedule them.

Due to the fact that an interpreter design using ADTs was chosen, the interpreter can simply be implemented as pattern matching on the opcodes. In fact, such an interpreter has been shown previously in Listing 21, namely the `lowLevelInterpreter` function. Here the function pattern matches on the effect `eff` and if its a `Success` opcode then it simply returns `Ok res` with `res` being the success value. More in-depth details of the implementation will be shown in Chapter 5.

4.2.2 Evaluation worker

The task of an evaluation worker is to schedule fibers and their effects for interpretation. This is based on a scheduling algorithm implemented by the particular runtime in use. An evaluation worker is designed as a separate thread that has access to a shared queue of effects that are ready to be interpreted. This is possible due to the design decisions made during the interpreter structure.

As all effects have the same type of `FIO<obj, obj>` internally, they can all be put into the same queue. This would not be possible if the effects had their specific types. The general idea is, instead of using OS threads for concurrency as with the naive interpreter, an improvement is to have a small amount of evaluation workers that schedule the fibers. This will improve performance as a constant amount of OS threads are spawned compared to arbitrary amount with the naive interpreter. In addition, the fibers will not be using as many resources as the OS threads.

The number of evaluation workers a runtime spawns is decided by the user when

the runtime is instantiated. To make sure that the work is spread out among all the evaluation workers, each evaluation worker will put an effect back into the queue after performing N interpretation steps (also known as evaluation steps). One evaluation step equals to interpretation of one opcode. This is to give the other workers a chance to perform some work as well, rather than having a single worker interpret a whole effect while the others are idle. This also makes switching between interpreting effects waiting in the queue faster, such that each waiting effect is interpreted bit by bit. The number of evaluation steps is decided by the user as well.

To provide better intuition about the evaluation worker, consider the following step by step success scenario using two evaluation workers.

Step 1 A runtime with 2 evaluation workers and 5 evaluation steps each is instantiated.

Step 2 The runtime is given an effect composed of 7 opcodes (meaning that it will take a worker 7 evaluation steps to complete interpretation). The effect is coupled with a fiber and added to the shared queue, then either of the two evaluation workers starts interpreting the effect.

Step 3 During interpretation, the worker will have used up its 5 evaluation steps at some point in time. Once that point is reached, it puts the remaining effect back into the queue together with the fiber to give the other worker a chance to work. The remaining effect now requires 2 evaluation steps before its completed.

Step 4 Either of the two evaluation workers will take the effect and fiber from the queue, depending on which thread is faster, and the worker will interpret the effect until its completed.

Step 5 Once the effect is completed, the resulting value of the effect is used to complete the fiber, and the result will be returned if the fiber is awaited. The shared queue is now empty and the workers are idle.

This can be seen as a general outline as to how the evaluation workers will be used. Precise details depends on the interpreter implementation. Currently, with the design of the evaluation workers, it is possible to use fibers as green threads when the amount of evaluation workers are greater than the amount of spawned fibers. Consider the scenario when a blocking opcode is interpreted waiting for a fiber or channel to retrieve data. What will happen currently is that the evaluation worker will be blocked until data is retrieved. This will cause deadlocks if only a single evaluation worker is used and is in general not very efficient. This can be solved by introducing another type of worker – the blocking worker.

4.2.3 Blocking worker

The use of a blocking worker will make it possible to use any number of evaluation workers and remove the previously mentioned deadlock issue. It reduces unnecessary wait time when evaluation workers interpret blocking opcodes as well.

The blocking worker achieves this by assisting the evaluation workers when its either waiting for a channel or fiber to retrieve data. The idea is, that when an evaluation worker has reached a blocking opcode of some effect, rather than waiting until data is received, the evaluation worker reschedules the whole effect to the blocking worker. This allows the evaluation worker to take a new fiber and effect from the shared queue to work on immediately with no wasted time in between. This also solves the deadlock, as the evaluation worker would now be able to interpret other effects that eventually could provide the blocked effect with data. Like the evaluation worker, the blocking worker is designed as an OS thread. Only a single blocking worker is required.

When an evaluation worker reschedules a blocking effect, the effect is added to a queue of blocking effects that is maintained by the blocking worker. It is then the task of the blocking worker to reschedule these effects back to the queue of the evaluation workers when an effect is not blocking anymore. This can be achieved in a variety of ways, however the simplest strategy would be the blocking worker checking its queue linearly. The blocking worker takes an effect from the blocking queue and checks if its still blocked. If it is, the effect is put back at the end of the queue, if its not, then it is put back into the shared queue of the evaluation workers as it is now ready to be interpreted. It then checks the next effect of the queue and so forth. Then the worker can check if any effect in the blocking queue is currently waiting for the given fiber or channel. If yes, then its known the effect is not blocked anymore as data has been retrieved.

4.2.4 Intermediate interpreter

The intermediate interpreter improves upon the naive interpreter by scheduling fibers as green threads. This is achieved by using simple designs of the evaluation and blocking workers. The evaluation worker will operate precisely as described in Section 4.2.2 and the blocking worker will use the strategy of linearly checking its queue as explained in Section 4.2.3.

As previously mentioned, this will improve performance by spawning a constant number of OS threads and having the evaluation workers ensure that work is

spread out among those threads. The blocking worker removes waiting time for blocked effects for the evaluation workers which will speed up interpretation as well.

It should be noted, that the intermediate interpreter is likely going to perform worse than the naive in scenarios with low concurrency and a lot of blocking. This is due to overhead of the workers and the linear checking of the blocked fibers. It may scale better than the naive when it comes to highly concurrent programs that does not include a lot of blocking.

4.2.5 Advanced interpreter

The advanced interpreter tries to improve the design of the intermediate interpreter by optimizing the way that blocked effects are handled.

Currently, the blocking worker of the intermediate interpreter checks the blocked effects in a linear manner which is not efficient. Consider a scenario where an effect is checked by the blocking worker while it is still blocked, so the effect is put back at the end of the blocking queue. At the same moment, it could happen that the effect retrieves some data and is therefore not blocked anymore. Even though the effect is not blocked, it will only be checked after all the effects in front of it have been checked. If the queue is sufficiently large, this will have an impact on performance. A better approach would be to reschedule the effect the moment it retrieves data – in constant time rather than linear.

This can be accomplished by instead of having a queue of blocked effects, let the fibers and channels contain the effects that are waiting on them. For fibers, it is then possible to reschedule all its waiting effects immediately after it is completed. This completely removes the necessity of the blocking worker for fibers, however, for channels, the blocking worker is still necessary. Whenever a channel receives some data, an event is sent to the blocking worker, letting the blocking worker know that the channel has received data. The blocking worker then removes one of the waiting effects contained in the channel and reschedules it back to the evaluation workers. This is the reason why a specific opcode, `SendMessage`, for sending a message to a channel is required. Otherwise it could have been implemented as an API function with the `NonBlocking` opcode.

It is expected that this design will perform better than the intermediate design in concurrent contexts with large amounts of blocking as effects that are no longer blocked are being reacted upon instantly.

4.2.6 Debugging tools

It is a well known fact that debugging race-conditions and deadlocks in concurrent programs is not an easy task. For this reason, 2 debugging tools will be designed to help combat potential programming errors along the way.

The tools will be implemented using conditional compilation symbols such that the debugging code will only be compiled when needed. This is to make sure that any of the additional code will not have an impact on performance. 2 tools will be created, a data structure monitor and a deadlock detector.

Data structure monitor The data structure monitor is designed as a thread that has access to critical data structures during runtime. The idea is for the monitor to present the contents of the data structures continuously to the user at a user defined rate. For example, if a deadlock is present in a specific scenario, FIO could be compiled with data structure monitoring support and the user could examine their contents. The most evident data structures to monitor would be the queues of the evaluation and blocking workers.

One may question whether a tool like this is necessary, as debugging tools in most development environments should be able to monitor the data as well. There is however, one key difference, which is that when using the data structure monitor, the program gets to run without being paused by breakpoints, let it be either in debug or release mode. It could happen, that pausing at specific breakpoints would prevent a deadlock from happening.

Deadlock detector The task of the deadlock detector is to keep track of whether a set of conditions that would cause a deadlock, all hold true at once. For example, if the evaluation workers queue is empty, and no evaluation worker is currently interpreting any effect, but there is at least one channel or fiber waiting for some data. If these conditions all hold true, a deadlock is certain as the waiting channel or fiber will never retrieve any data. As with the data structure monitor, the deadlock detector would be implemented as a thread as well.

4.3 Summary

In this chapter, the design of FIO was split up into two parts, the effect and the runtime system. For the effect system, the effect API was designed with a multitude of API functions as seen in table Table 4.1. The *O2* objective

was deemed satisfied by the functions. In addition, a challenge was introduced in regards of balancing how much logic should be kept inside the API versus opcodes. Furthermore, the effect type of FIO was designed to be `FIO<'R, 'E>` with success type `'R` and error type `'E`.

Moreover, several challenges in the context of interpreter structure design was discussed. This included problems with missing GADT support in F# and the requirement of having explicit types for type parameters when pattern matching on classes. In addition to that, the visitor pattern was suggested by Tomas Petricek and an approach using ADTs with type casting by Alceste. These two design patterns were presented alongside their advantages and disadvantages and at the end, the latter approach was chosen. This was due to the design having a more simple implementation and using `FIO<obj, obj>` would make it easier to implement advanced interpretation techniques. Next, the concept of channels and fibers were formally introduced and how they would be designed with the chosen interpreter structure. Then, the designed opcodes were shown in Table 4.2 after their design challenges were discussed in further detail.

For the runtime system, the naive interpreter was designed as a baseline interpreter using OS threads to be compared against the other interpreters. The evaluation worker was then introduced with having the task of scheduling fibers as green threads. The blocking worker was introduced as well with the job of assisting the evaluation worker in handling blocking effects to improve performance and reduce unnecessary wait time when interpreting blocked effects. Then the intermediate interpreter was introduced to improve upon the naive interpreter by using simple designs of the evaluation and blocking workers. In this interpreter, the blocking worker checks whether the effects are blocking in a linear manner which is not efficient. Furthermore, the advanced interpreter was introduced to optimize the intermediate interpreter by changing the linear checking of the blocking worker into a constant time check instead. At the end, two debugging tools was designed to assist with the implementation of the interpreter logic. A data structure monitor that will show the user contents of important data structures during runtime, and a deadlock detector that attempts to detect deadlocks during interpretation.

In the next chapter, more specific implementation details of some of the elements discussed in the design will take place.

Chapter 5

Implementation

This chapter will elaborate further upon the technical details that were introduced in Chapter 4. First and foremost, implementation details will be presented for the effect system, including further insight into the effect API, interpreter structure, channels and fibers. Next, implementation details will be shown for the runtime system, including how the naive, intermediate and advanced interpreters are implemented together with evaluation and blocking workers. Finally, the debugging tools of the data structure monitor and deadlock detector will have their implementations presented as well.

5.1 Effect system

As the effect and runtime system have been designed, it is feasible to start introducing technical implementation details of the effect API and related concepts.

5.1.1 Effect API

As explained in Section 4.1.1 various API functions are available as shown in Table 4.1. To show how API functions are implemented, the implementation of the most commonly used functions such as `spawn`, `await`, >>, ||| and |||* is presented. The remaining API functions can be found in Appendix A.

As mentioned previously, the `spawn` function is merely an alias for the `Concurrent` opcode with the extra step of instantiating a fiber. Consider its implementation as shown in Listing 25. The function takes 3 type parameters as the errors of the fiber and effect may be different. Notice that the returned effect of `FIO<Fiber<'R1, 'E1>, 'E>` is an effect returning a fiber, that is the result of this function is the spawned fiber. On line 2, the fiber is instantiated with appropriate type parameters and the `Concurrent` opcode is returned with an upcasted effect, the fiber and its corresponding low-level fiber. This is sufficient to hide the type casting from the user.

```
1 let spawn<'R1, 'E1, 'E> (eff : FIO<'R1, 'E1>) :
  ↪ FIO<Fiber<'R1, 'E1>, 'E> =
2   let fiber = new Fiber<'R1, 'E1>()
3   Concurrent (eff.Upcast(), fiber, fiber.ToLowLevel())
```

Listing 25: `spawn` API function implementation

A function that is commonly used in cooperation with `spawn` is that of `await`. The implementation of `await` can be found in Listing 26. This function is close to an alias as it simply converts to the fiber to a low-level fiber and passes it as an argument to the `AwaitFiber` opcode.

```
1 let await<'R, 'E> (fiber : Fiber<'R, 'E>) : FIO<'R, 'E> =
2   AwaitFiber <| fiber.ToLowLevel()
```

Listing 26: `await` API function implementation

The next implementation is that of >>, the infix sequencing function as shown in Listing 27. Parentheses are required around the function name to turn it infix. This function simply uses the `SequenceSuccess` opcode, passes it an upcasted effect and a continuation where the argument is upcasted to `'R1`. This is required as the argument (`res`) of the opcodes continuation function is of type `obj` as seen in Table 4.2 (`(obj -> FIO<'R, 'E>)`). However, as the result of the sequencing is available to the user, it is required that a concrete type is used as shown in the continuation that is passed to the API function itself (`(cont : 'R1 -> FIO<'R, 'E>)`). It should be noted, that its not strictly necessary that this function is infix, however, making it infix provides a nicer and more readable syntax when sequencing large amounts of effects.

```
1  let (>>) (eff : FIO<'R1, 'E>) (cont : 'R1 -> FIO<'R, 'E>) :
   ↪ FIO<'R, 'E> =
2    SequenceSuccess (eff.UpcastResult(), fun res -> cont
     ↪ (res :?> 'R1))
```

Listing 27: >> infix API function implementation

Another widely used function is |||, the parallel operator as shown in Listing 28. This infix function takes two effects as arguments with the success types of `'R1` and `'R2` respectively. The resulting effect of `FIO<'R1 * 'R2, 'E>` may succeed with a tuple of the two results or fail with some error `'E`. This function is the first example that is composed of other API functions, namely `spawn`, `await` and >>. ||| spawns a single fiber to interpret `eff1` concurrently and lets the other effect `eff2` interpret on the current fiber. The result of the fiber is awaited and the results are passed to the `Success` opcode as a tuple. As with >>, this function is made infix to provide a clean syntax.

```
1  let (|||) (eff1 : FIO<'R1, 'E>) (eff2 : FIO<'R2, 'E>) :
   ↪ FIO<'R1 * 'R2, 'E> =
2    spawn eff1 >> fun fiber1 ->
3    eff2 >> fun res2 ->
4    await fiber1 >> fun res1 ->
5    Success (res1, res2)
```

Listing 28: ||| infix API function implementation

The implementation of the |||∗ function can be found in Listing 29. This has the same functionality as |||, but instead of returning a tuple of the results it returns `Unit`. This function was created as it was found to be a common pattern

that ||| would be used to interpret effects in parallel and the results would be discarded. Creating this function makes the code easier to understand rather than having to use ||| and discard the results manually.

```
let (|||*) (eff1 : FIO<'R1, 'E>) (eff2 : FIO<'R2, 'E>) :
↪  FIO<Unit, 'E> =
  eff1 ||| eff2
  >> fun (_, _) ->
  stop
```

Listing 29: |||* infix API function implementation

With the functions being implemented in less than 10 lines of code, it shows that they are simple and describes themselves which further supports *O2*. Extending FIO with further functionality would be a straightforward procedure as well.

5.1.2 Interpreter structure

An example of using ADTs with type casting for the interpreter structure to implement two opcodes – `Success` and `SequenceSuccess` – was shown in Listing 21 in Section 4.1.3.2. Consider the final implementation of this pattern as shown in Listing 30, albeit with no interpreter present.

A few important implementation details are present. For example, notice that on line 1, the cases (opcodes) of the discriminated union `FIO` are declared internal. This is to ensure that the opcodes are not visible to the user. This is the same reason for the `UpcastResult`, `UpcastError` and `Upcast` functions as they are for internal use only as well. It should be emphasized, that the `FIO<'R, 'E>` type is available to the user, it is only the cases of the type that are not. In other words, it is the `FIO<'R, 'E>` type and the API functions that are visible to the user. Recall that in the previous section, the API functions had no explicit control specifier, meaning that they are all public per default.

The type casting functions of `UpcastResult`, `UpcastError` and `Upcast` are implemented as member functions of the `FIO` type and can be called using the `.` operator as seen on line 35. Examining the implementation of `UpcastResult`, it is simply a pattern match on all possible cases of `FIO` where the same type is returned however with the result upcasted to `obj`. An example of this can be seen on line 29, where the the `res` value of `Success` is upcasted. In addition, even though the `Upcast` function is not strictly required, it is implemented as sometimes it is needed to upcast both the success and error type, and calling a single

function is simpler than calling two in succession. The code for `UpcastError` is not shown as it is similar to that of `UpcastResult`.

```
1  type FIO<'R, 'E> = internal
2    | NonBlocking of action: (unit -> Result<'R, 'E>)
3    | Blocking of chan: Channel<'R>
4    | SendMessage of msg: 'R * chan: Channel<'R>
5    | Concurrent of effect: FIO<obj, obj> * fiber: obj * llfiber:
     ↪  LowLevelFiber
6    | AwaitFiber of llfiber: LowLevelFiber
7    | SequenceSuccess of effect: FIO<obj, 'E> * cont: (obj ->
     ↪  FIO<'R, 'E>)
8    | SequenceError of FIO<obj, obj> * cont: (obj -> FIO<'R, 'E>)
9    | Success of result: 'R
10   | Failure of error: 'E
11
12   member internal this.UpcastResult<'R, 'E>() : FIO<obj, 'E> =
13     match this with
14     | NonBlocking action ->
15         NonBlocking <| fun () ->
16         match action () with
17           | Ok res -> Ok (res :> obj)
18           | Error err -> Error err
19     | Blocking chan -> Blocking <| chan.Upcast()
20     | SendMessage (msg, chan) ->
21         SendMessage (msg :> obj, chan.Upcast())
22     | Concurrent (eff, fiber, llfiber) ->
23         Concurrent (eff, fiber, llfiber)
24     | AwaitFiber llfiber -> AwaitFiber llfiber
25     | SequenceSuccess (eff, cont) ->
26         SequenceSuccess (eff, fun res -> (cont
           ↪  res).UpcastResult())
27     | SequenceError (eff, cont) ->
28         SequenceError (eff, fun res -> (cont res).UpcastResult())
29     | Success res -> Success (res :> obj)
30     | Failure err -> Failure err
31
32   member internal this.UpcastError<'R, 'E>() : FIO<'R, obj> =
     ↪  // logic <snip>
33
34   member internal this.Upcast<'R, 'E>() : FIO<obj, obj> =
35     this.UpcastResult().UpcastError()
```

Listing 30: Interpreter structure implementation using ADTs with type-casting

5.1.3 Channels

In Section 4.1.4.1 the concept of a channel was introduced and a simple implementation example was shown in Listing 22. The final implementation as seen in Listing 31 has retrieved extra functions in cooperation with the intermediate and advanced interpreters.

To elaborate more in-depth, a channel is a type that is wrapped around a `BlockingCollection<obj>`. `BlockingCollection<obj>` is a thread-safe queue that store values of the type `obj`. Using this collection type helps make channels thread-safe for concurrent workflows with no extra implementation. Notice that the `BlockingCollection<obj>` named `chan` is used with the `Add` and `Take` functions, that is `chan` is the actual data structure that stores data of a channel. Due to `chan` storing values of type `obj`, it is convenient for `Add` to specify the type of the value it receives as `'R` and required for `Take` to explicitly return `'R`. The reason why it was decided to implement channels as a type that wraps a `BlockingCollection<obj>` is because it allows for controlling which functionality the type has.

The channel constructor on line 1 has been declared private as it should not be possible for the user to create a channel and pass it a `BlockingCollectio<obj>`. The constructor is only for internal usage with the `Upcast` function, and only the `new` operator should be used to create a channel which ensures that an empty `BlockingCollectio<obj>` is used. Another element worth noticing is that it is only the `Add`, `Take` and `Count` functions that are public, as these are the only functions the user would need.

`blockingWorkItems` is the queue that stores blocked effects that are waiting on the channel. A work item is simply a pair of an effect and a fiber that will hold the result of the effect. It is used exclusively with the advanced interpreter as described in Section 4.2.5. When an effect is waiting on the channel, the `AddBlockingWorkItem` function is used to add a work item to the queue. When data has been received, the `RescheduleBlockingWorkItem` function is used to remove the head of the queue.

The `dataCounter` value is used to fix a race-condition that was introduced with the intermediate interpreter. This value counts the number of available data in the channel, which may or may not be equal to the amount of data present in `chan`. The Interlocked .NET API is used to increment and decrement the value atomically to avoid data races. The race-condition will be explained further in Section 5.2.2.2.

```
type Channel<'R> private (
  chan: BlockingCollection<obj>,
  blockingWorkItems: BlockingCollection<WorkItem>,
  dataCounter: int64 ref) =

  new() = Channel(new BlockingCollection<obj>(),
                  new BlockingCollection<WorkItem>(),
                  ref 0)

  member internal _.AddBlockingWorkItem workItem =
    blockingWorkItems.Add workItem

  member internal _.RescheduleBlockingWorkItem
  ↪  (workItemQueue: BlockingCollection<WorkItem>) =
    if blockingWorkItems.Count > 0 then
      workItemQueue.Add <| blockingWorkItems.Take()

  member internal _.HasBlockingWorkItems() =
    blockingWorkItems.Count > 0

  member internal _.Upcast() =
    Channel<obj>(chan, blockingWorkItems, dataCounter)

  member internal _.UseAvailableData() =
    Interlocked.Decrement dataCounter |> ignore

  member internal _.DataAvailable() =
    Interlocked.Read dataCounter > 0

  member _.Add (value : 'R) =
    Interlocked.Increment dataCounter |> ignore
    chan.Add value

  member _.Take() : 'R = chan.Take() :?> 'R

  member _.Count() = chan.Count
```

Listing 31: `Channel` type implementation

5.1.4 Fibers

In a similar fashion to channels, the fibers are implemented as types that wrap around a BlockingCollection<Result<obj, obj». In the case of fibers, the reason for this is to provide a thread-safe construct that can be awaited. The idea is for the BlockingCollection<Result<obj, obj» named chan to hold a result of the type Result<obj, obj> that it is completed by some interpretation. 3 important rules that fibers must adhere to were presented in Section 4.1.4.2. Namely, **R1**: A fiber may only be completed once, **R2**: If a fiber is being completed concurrently, only one of the completions may be applied, and **R3**: If a completed fiber is awaited, it will always return the same result. These rules are necessary to be able to have correct programs in FIO.

Consider the implementation of the LowLevelFiber as seen in Listing 32. **R1** is satisfied by using the completed value as a guard. If completed is equal to 0 the fiber has not been completed, if its equal to 1 it has been completed. An exception is thrown on line 11 if the fiber is attempted to be completed more than once. This is safe, as FIO's runtime system should ensure that a LowLevelFiber is never completed more than once, thus if it happens, it indicates an implementation error in the runtime system.

The second rule of **R2** is ensured by using the Interlocked API to make reading and writing to completed atomic operations as seen on line 7 and 8. This way, only the first call to the Complete function will be applied as once a second call is allowed completed will have changed to 1 and an exception will be thrown.

When a fiber is awaited, a result is tried to be retrieved from chan as seen on line 14. Since a BlockingCollection is used, if chan is empty it will simply block the caller, however if its not, the first value in the queue will be returned. Next, on line 15, the value is immediately added back to chan and then the value is returned to the caller. This way, there is always a single value residing in chan no matter how many times the fiber is awaited concurrently which satisfies the last rule of **R3**. This approach minimizes performance loss as the value is added back immediately. It could be the case, that if a fiber is awaited concurrently in extreme amounts, that some performance loss may be noticeable.

As with channels, fibers keep track of the effects that are waiting on them when using the advanced interpreter. For this reason, a LowLevelFiber has access to a BlockingCollection<WorkItem> as well. The rescheduling function RescheduleBlockingWorkItems is a slightly different than the one found in channels, as this one reschedules all effects that are waiting instead of just one.

```
type internal LowLevelFiber internal (
  chan: BlockingCollection<Result<obj, obj>>,
  blockingWorkItems: BlockingCollection<WorkItem>,
  completed: int64 ref) =

  member internal _.Complete res =
    if Interlocked.Read completed = 0 then
      Interlocked.Exchange(completed, 1) |> ignore
      chan.Add res
    else
      failwith "LowLevelFiber: Complete was called on an already
      ↪  completed LowLevelFiber!"

  member internal _.Await() =
    let res = chan.Take()
    chan.Add res
    res

  member internal _.Completed() =
    Interlocked.Read completed = 1

  member internal _.AddBlockingWorkItem workItem =
    blockingWorkItems.Add workItem

  member internal _.BlockingWorkItemsCount() =
    blockingWorkItems.Count

  member internal _.RescheduleBlockingWorkItems
  ↪  (workItemQueue : BlockingCollection<WorkItem>) =
    while blockingWorkItems.Count > 0 do
      workItemQueue.Add <| blockingWorkItems.Take()
```

Listing 32: `LowLevelFiber` type implementation

Switching focus to the fiber handle, its implementation can be found in Listing 33. The fiber handle does not include much functionality as it is just a handle to a `LowLevelFiber`.

As mentioned previously, a `Fiber` and its corresponding `LowLevelFiber` have references to the same data, that is the same instance of `chan`, `blockingWorkItems` and `completed`. In fact, previous errors were present because the state of `completed` was not shared between a `Fiber` and any `LowLevelFiber` that was converted from that. This caused the `Fiber` to indicate it was not completed when its corresponding `LowLevelFiber` would indicate that it was completed. This error was caught by the data structure monitor which implementation will

be discussed later. Since the `Fiber` handle may be awaited by the user of the library, it requires to have a similar `await` function to that of `LowLevelFiber`, however with the difference that the result is downcasted to its specific values.

```
1  type Fiber<'R, 'E> private (
2    chan: BlockingCollection<Result<obj, obj>>,
3    blockingWorkItems: BlockingCollection<WorkItem>) =
4    let completed : int64 ref = ref 0
5
6    new() = Fiber(new BlockingCollection<Result<obj, obj>>(),
7                  new BlockingCollection<WorkItem>())
8
9    member internal _.ToLowLevel() =
10     LowLevelFiber(chan, blockingWorkItems, completed)
11
12   member _.Await() : Result<'R, 'E> =
13     let res = chan.Take()
14     chan.Add res
15     match res with
16     | Ok res -> Ok (res :?> 'R)
17     | Error err -> Error (err :?> 'E)
```

Listing 33: `Fiber` type implementation

5.2 Runtime system

In Chapter 2 it was specified that the interpreters are able to spawn any desired number of fibers. This is because the interpreters are tail-call optimized which means that allocation of a new stack frame at every recursive call is avoided, and thus they use constant stack space.

5.2.1 Naive interpreter

An interpreter makes use of 3 auxiliary handler functions as shown in Listing 34. One for handling a success result called `handleSuccess` and one for failure results called `handleError`. The third function called `handleResult` handles all results by using pattern matching to check whether it is a success or error and then invokes the appropriate handler function.

Due to the tail-call optimization of the interpreter functions, the functions no longer allocate stack frames. This causes an issue in the context of sequencing effects, that is using the `SequenceError` and `SequenceSuccess` opcodes. A stack frame is required for the interpreter to know when to use a continuation created by the opcodes. The task of the handler functions is to maintain a virtual stack frame such that the interpreter still knows when to use available continuations. They do this by accepting a list of the `StackFrame` type – the virtual stack – such that they can check if any continuations are available. Whenever a `SequenceSuccess` opcode is interpreted its continuation function is added to the list as a `SuccHandler` type and the same with `SequenceError` and `ErrorHandler`. For example, in the case of `handleSuccess`, if the virtual stack frame `stack` is empty, the `Ok` result is returned immediately. However, if its not, the function recursively attempts to find the first `SuccHandler` such that the result can be applied to it. Since the context is a success scenario, any `ErrorHandler` present in the stack can safely be ignored.

```
1  type internal StackFrame =
2    | SuccHandler of succCont: (obj -> FIO<obj, obj>)
3    | ErrorHandler of errCont: (obj -> FIO<obj, obj>)
4
5  let rec handleSuccess res stack =
6    match stack with
7    | [] -> Ok res
8    | s::ss -> match s with
9               | SuccHandler succCont ->
10                  this.LowLevelRun (succCont res) ss
11              | ErrorHandler _ ->
12                  handleSuccess res ss
13
14 let rec handleError err stack =
15   match stack with
16   | [] -> Error err
17   | s::ss -> match s with
18              | SuccHandler _ ->
19                  handleError err ss
20              | ErrorHandler errCont ->
21                  this.LowLevelRun (errCont err) ss
22
23 let handleResult result stack =
24   match result with
25   | Ok res -> handleSuccess res stack
26   | Error err -> handleError err stack
```

Listing 34: Implementation of the naive interpreter's handler functions

As mentioned in Section 4.1.3.2, the chosen design pattern of algebraic data types with type casting requires two interpretation functions, one that accepts `FIO<obj, obj>` and another for `FIO<'R, 'E>`. The implementation of these two functions for the naive interpreter can be found in Listing 35 as `LowLevelRun` and `Run` respectively. As this is the naive interpreter, most of the interpretation logic is fairly simple. A few highlights include the previously mentioned continuations of `SequenceSuccess` and `SequenceError` that are added to `stack` as seen on line 15 and 17. In addition, for the `Concurrent` opcode, async computation expressions are used to spawn a new OS thread as seen on line 10. For the `AwaitFiber` opcode, the interpreter awaits the result and it is passed along to `handleResult`. For the `Run` function, the passed effect `eff` is interpreted in a new OS thread which passes the result to a fiber handle that is returned to the user.

```
1   member internal this.LowLevelRun eff (stack : List<StackFrame>)
2     : Result<obj, obj> =
3     match eff with
4     | NonBlocking action -> handleResult (action ()) stack
5     | Blocking chan -> let res = chan.Take()
6                        handleSuccess res stack
7     | SendMessage (value, chan) -> chan.Add value
8                                    handleSuccess value stack
9     | Concurrent (eff, fiber, llfiber) ->
10        async { llfiber.Complete <| this.LowLevelRun eff [] }
11        |> Async.StartAsTask |> ignore
12        handleSuccess fiber stack
13    | AwaitFiber llfiber -> handleResult (llfiber.Await()) stack
14    | SequenceSuccess (eff, cont) ->
15        this.LowLevelRun eff (SuccHandler cont :: stack)
16    | SequenceError (eff, cont) ->
17        this.LowLevelRun eff (ErrorHandler cont :: stack)
18    | Success res -> handleSuccess res stack
19    | Failure err -> handleError err stack
20
21  override this.Run<'R, 'E> (eff : FIO<'R, 'E>) : Fiber<'R, 'E> =
22    let fiber = new Fiber<'R, 'E>()
23    async { fiber.ToLowLevel().Complete <| this.LowLevelRun
      ↪  (eff.Upcast()) [] }
24    |> Async.StartAsTask |> ignore
25    fiber
```

Listing 35: Implementation of the naive interpreter's `LowLevelRun` and `Run` functions

5.2.2 Intermediate interpreter

The intermediate interpreter uses the same auxiliary functions as the naive interpreter as seen in Listing 36. In fact, the same handler functions are used by both the intermediate and advanced interpreters. The handler functions are almost identical to the ones of the naive interpreter, with the difference that instead of only returning the result, more information is provided. For example, at line 8 when no virtual stack frames are present, a triple consisting of a tuple of the result and current stack, the next action to do with the result, and the new evaluation steps is returned. In this specific case a `Success` opcode with an empty stack is returned to the evaluation worker together with the `Evaluated` action.

The next action of an effect is determined by the `Action` type as defined on line 1. It is one out of 3 cases, namely `RescheduleForRunning`, `RescheduleForBlocking` or `Evaluated`. `RescheduleForRunning` tells an evaluation worker that the given effect should be put back into the work item queue to be interpreted later. This is used in the `LowLevelRun` function when the evaluation steps has reached 0. `RescheduleForBlocking` signifies that the effect is blocking and should thus be rescheduled to the blocking worker. The type contains the blocking opcode. Precisely how these actions are used by the evaluation workers will be explained in their respective sections later. Lastly, `Evaluated` simply means that the effect can be evaluated safely as is the case with the `handleSuccess` function.

Before the interpretation functions are presented, it is required to introduce the implementation of the work item. As previously mentioned during this chapter, a work item is the unit of work that the evaluation worker works with and consists of an effect and its corresponding fiber as seen in Listing 37. One additional detail, is that it also contains the associated virtual stack frame for the effect. This used when the interpreter meets a blocking effect. When completing a work item, it simply means that its fiber is completed.

```
1  type internal Action =
2    | RescheduleForRunning
3    | RescheduleForBlocking of BlockingItem
4    | Evaluated
5
6  let rec handleSuccess res newEvalSteps stack =
7    match stack with
8    | [] -> ((Success res, []), Evaluated, newEvalSteps)
9    | s::ss -> match s with
10               | SuccHandler succCont ->
11                   this.LowLevelRun (succCont res) Evaluated
                     ↪ evalSteps ss
12               | ErrorHandler _ ->
13                   handleSuccess res newEvalSteps ss
14
15 let rec handleError err newEvalSteps stack =
16   match stack with
17   | [] -> ((Failure err, []), Evaluated, newEvalSteps)
18   | s::ss -> match s with
19              | SuccHandler _ ->
20                  handleError err newEvalSteps ss
21              | ErrorHandler errCont ->
22                  this.LowLevelRun (errCont err) Evaluated
                    ↪ evalSteps ss
23
24 let handleResult result newEvalSteps stack =
25   match result with
26   | Ok res -> handleSuccess res newEvalSteps stack
27   | Error err -> handleError err newEvalSteps stack
```

Listing 36: Implementation of the intermediate and advanced interpreter's handler functions and `Action` type

```
1  type internal WorkItem =
2    { Eff: FIO<obj, obj>; Stack: List<StackFrame>; LLFiber:
     ↪ LowLevelFiber; PrevAction: Action }
3    static member Create eff stack llfiber prevAction =
4      { Eff = eff; Stack = stack; LLFiber = llfiber; PrevAction =
       ↪ prevAction }
5    member this.Complete res =
6      this.LLFiber.Complete <| res
```

Listing 37: `WorkItem` type implementation

On Listing 38 the implementation of the interpretation functions is shown. The primary difference between this and the naive interpreter is how blocking and concurrent effects are handled. Consider the if-expression on line 12. The first time a `Blocking` opcode is met, it is assumed that it is blocking and thus the effect is returned with the current virtual stack and `RescheduleForBlocking` with the blocking channel as argument. For this reason, it is known that whenever the previous action `prevAction` is `RescheduleForBlocking`, the effect has been scheduled back by the blocking worker and data can now safely be retrieved from the channel.

A similar situation is present when the `AwaitFiber` opcode is interpreted. Examining line 25, it is directly checked whether the fiber is completed. One may wonder why the channel is not checked directly. The reason for this is a fundamental difference between channels and fibers. As **R1** and **R3** recites for fibers, a fiber may only be completed once and the same result will always be returned. This is not the case for channels as they might be empty even if they have contained data previously. Due to this, if the channel was checked directly, it could lead to race-conditions if the channel is awaited concurrently. The solution to this problem will be explained later. Moving on from line 25, it is safe to await the fiber if it is already completed. If not, the fiber is rescheduled for blocking by returning the effect and its virtual stack frames with the `RescheduleForBlocking` action.

For the `Concurrent` opcode on line 22, a new instance of the `WorkItem` type is added to the `workItemQueue`. It simply passes the effect, an empty virtual stack frame and the fiber of the opcode into a new `WorkItem` and adds that to the queue so it can be interpreted by another or the same evaluation worker later.

Examining the `Run` function, it simply creates a `WorkItem` out of the passed effect, an empty stack frame and returns the fiber. This way, an evaluation worker will take the effect and start interpreting immediately.

```
member internal this.LowLevelRun eff prevAction evalSteps
  (stack : List<StackFrame>)
  : (FIO<obj, obj> * List<StackFrame>) * Action * int =
  if evalSteps = 0 then
    ((eff, stack), RescheduleForRunning, 0)
  else
    let newEvalSteps = evalSteps - 1
    match eff with
    | NonBlocking action ->
        handleResult (action ()) newEvalSteps stack
    | Blocking chan ->
        if prevAction = RescheduleForBlocking (BlockingChannel
        ↪  chan) then
          let res = chan.Take()
          handleSuccess res newEvalSteps stack
        else
          ((Blocking chan, stack),
            RescheduleForBlocking (BlockingChannel chan),
            ↪  evalSteps)
    | SendMessage (value, chan) ->
        chan.Add value
        handleSuccess value newEvalSteps stack
    | Concurrent (eff, fiber, llfiber) ->
        workItemQueue.Add <| WorkItem.Create eff [] llfiber
        ↪  prevAction
        handleSuccess fiber newEvalSteps stack
    | AwaitFiber llfiber ->
        if llfiber.Completed() then
          handleResult (llfiber.Await()) newEvalSteps stack
        else
          ((AwaitFiber llfiber, stack),
            RescheduleForBlocking (BlockingFiber llfiber),
            ↪  evalSteps)
    | SequenceSuccess (eff, cont) ->
        this.LowLevelRun eff prevAction evalSteps (SuccHandler
        ↪  cont :: stack)
    | SequenceError (eff, cont) ->
        this.LowLevelRun eff prevAction evalSteps (ErrorHandler
        ↪  cont :: stack)
    | Success res -> handleSuccess res newEvalSteps stack
    | Failure err -> handleError err newEvalSteps stack

override _.Run<'R, 'E> (eff: FIO<'R, 'E>) : Fiber<'R, 'E> =
  let fiber = Fiber<'R, 'E>()
  workItemQueue.Add <| WorkItem.Create (eff.Upcast()) []
  ↪  (fiber.ToLowLevel()) Evaluated
  fiber
```

Listing 38: Implementation of the intermediate interpreter's `LowLevelRun` and `Run` functions

5.2.2.1 Evaluation worker

The job of the evaluation worker is to interpret the `WorkItem` instances in the `workItemQueue`. Its implementation can be seen on Listing 39. On line 10, an async computation expression is used to run the logic of the worker in a new thread. Furthermore, on line 11 the worker continuously checks for new work items in the queue and interprets them. Once an interpretation is completed, the evaluation worker will take appropriate action depending on the returned `Action` type.

Evaluated If the `Evaluated` action is returned, it is known that the effect has been evaluated and is therefore returned with either a `Success` or a `Failure` opcode. This means that the evaluation worker should complete the current work item with the approppriate result – either `Ok` or `Error` as seen on line 17 and 19. The stack frame returned together with effects that are `Evaluated` are always empty and are therefore discarded.

RescheduleForRunning When this action is returned, the worker simply creates a new work item with the remaining effect that is returned and its fiber and puts it into the `workItemQueue` to make it available for further interpretation. As continuations may be available in the returned stack frame, the stack frame is passed to the new work item as well.

RescheduleForBlocking When the `RescheduleForBlocking` action is returned a new work item is created with the blocking effect and the action. The work item is then passed further to the blocking worker. The stack frame is passed to the work item as continuations may be available here as well.

The compilation directives seen on line 5 and 8, 12 and 14 and so forth guard logic that is only necessary to be compiled when compiling with deadlock detection support. This is to satisfy the requirement stated in Section 4.2.6.

```
type internal EvalWorker(runtime: Runtime,
  workItemQueue: BlockingCollection<WorkItem>,
  blockingWorker: BlockingWorker, evalSteps) =

  #if DETECT_DEADLOCK
  inherit Worker()
  let mutable working = false
  #endif

  let _ = (async {
    for workItem in workItemQueue.GetConsumingEnumerable() do
      #if DETECT_DEADLOCK
      working <- true
      #endif
      match runtime.LowLevelRun workItem.Eff workItem.PrevAction
      ↪  evalSteps workItem.Stack with
      | (Success res, _), Evaluated, _ ->
          workItem.Complete (Ok res)
      | (Failure err, _), Evaluated, _ ->
          workItem.Complete (Error err)
      | (eff, stack), RescheduleForRunning, _ ->
          let workItem = WorkItem.Create eff stack
          ↪  workItem.LLFiber RescheduleForRunning
          workItemQueue.Add workItem
      | (eff, stack), RescheduleForBlocking blockingItem, _ ->
          let workItem = WorkItem.Create eff stack
          ↪  workItem.LLFiber (RescheduleForBlocking
          ↪  blockingItem)
          blockingWorker.RescheduleForBlocking blockingItem
          ↪  workItem
      | _ -> failwith $"EvalWorker: Error occurred while
      ↪  evaluating effect!"

      #if DETECT_DEADLOCK
      working <- false
      #endif
    } |> Async.StartAsTask |> ignore)

#if DETECT_DEADLOCK
override _.Working() =
    working && workItemQueue.Count > 0
#endif
```

Listing 39: Implementation of the intermediate interpreter's evaluation worker

5.2.2.2 Blocking worker

The internal API of the blocking worker consists of the `ResheduleForBlocking` function as seen in Listing 40 on line 42. As shown in the previous section, this is the function that the evaluation worker uses to pass any blocking effects to the blocking worker. The function simply adds the `BlockingItem` – either a channel or a fiber – and the work item containing the blocking effect to its `blockingItemQueue`. On line 12, the worker iterates through all current blocking items and then uses the `HandleBlockingItem` function to check whether it is still blocking or not. For example, on line 25 it is checked whether a channel has data available and line 34 for fibers. It should be noted, that the `Working` function as seen for the evaluation worker exists for the blocking worker as well, however as been removed to simplify the listing.

A race-condition was previously mentioned in Section 5.2.2 in regards to channels. This race condition is the reason the `DataAvailable` and `UseAvailableData` functions were implemented for the channel type. The idea behind the functions were briefly discussed in Section 5.1.3. Imagine the following scenario. Two effects are being interpreted concurrently that both await data on the same channel. When the blocking worker sees that the channel has data available, it reschedules the corresponding effect for interpretation. Consider the scenario where the next blocking effect that is processed by the blocking worker is also waiting on data from the same channel. The previous effect may not yet have been interpreted, so the data is still available in the channel. This makes the blocking effect reschedule the second effect as well. This is a fatal flaw, because the channel may only contain a single element of data, but 2 elements are now needed. This could eventually cause a deadlock, because one of the effects may never retrieve their required data.

This is where the `DataAvailable` and `UseAvailableData` functions come into the place. These functions ensure that the data is "used" immediately such that the blocking worker may never reschedule an effect for running based on data that is already used for another effect. This way, the second effect in the queue would not have been scheduled back and thus the race-condition and potential deadlock issues are solved. In more detailed terms, on line 25 the blocking worker checks whether there is any data available. If yes, it uses this data immediately by decrement the internal `dataCounter` of the channel using `UseAvailableData`. It should be noted, that this problem scenario was caught using the data structure monitoring tool.

```
type internal BlockingWorker(
  workItemQueue: BlockingCollection<WorkItem>,
  #if DETECT_DEADLOCK
  deadlockDetector: Utils.DeadlockDetector<BlockingWorker,
  ↪  EvalWorker>,
  #endif
  blockingItemQueue: BlockingCollection<BlockingItem * WorkItem>)
  ↪  as self =
  #if DETECT_DEADLOCK
  inherit Worker()
  let mutable working = false
  #endif
  let _ = (async {
    for blockingItem, workItem in
    ↪  blockingItemQueue.GetConsumingEnumerable() do
      #if DETECT_DEADLOCK
      working <- true
      #endif
      self.HandleBlockingItem blockingItem workItem
      #if DETECT_DEADLOCK
      working <- false
      #endif
  } |> Async.StartAsTask |> ignore)

  member private _.HandleBlockingItem blockingItem workItem =
    match blockingItem with
    | BlockingChannel chan ->
        if chan.DataAvailable() then
          chan.UseAvailableData()
          workItemQueue.Add workItem
          #if DETECT_DEADLOCK
          deadlockDetector.RemoveBlockingItem blockingItem
          #endif
        else
          blockingItemQueue.Add ((blockingItem, workItem))
    | BlockingFiber llfiber ->
        if llfiber.Completed() then
          workItemQueue.Add workItem
          #if DETECT_DEADLOCK
          deadlockDetector.RemoveBlockingItem blockingItem
          #endif
        else
          blockingItemQueue.Add ((blockingItem, workItem))

  member internal _.RescheduleForBlocking blockingItem workItem =
    blockingItemQueue.Add ((blockingItem, workItem))
    #if DETECT_DEADLOCK
    deadlockDetector.AddBlockingItem blockingItem
    #endif
```

Listing 40: Implementation of the intermediate interpreter's blocking worker

5.2.3 Advanced interpreter

The advanced interpreter makes use of the same handler functions as the intermediate that was previously shown in Listing 36. The interpretation functions for the advanced interpreter are identical to the ones seen previously in Listing 38 however with one small change. As mentioned during the design in Section 4.2.5, whenever a message is sent through a channel, an event is sent to the blocking worker. Therefore, the only difference between the intermediate and advanced interpreter's `LowLevelRun` is the `SendMessage` opcode implementation which can be seen in Listing 41. A simple extra step is added, sending an event which is the channel itself to the worker's `blockingEventQueue`.

```
1  // logic <snip>
2  | SendMessage (value, chan) ->
3      chan.Add value
4      blockingEventQueue.Add <| chan
5      handleSuccess value newEvalSteps stack
6  // logic <snip>
```

Listing 41: Changes to the implementation of the advanced interpreter's `LowLevelRun` function

5.2.3.1 Evaluation worker

The implementation of the evaluation worker used for the advanced interpreter is presented in Listing 42. Deadlock detector compilation symbols have been removed as they are very similar to the ones found in the evaluation worker for the intermediate interpreter. The differences from the intermediate's evaluation worker include the `CompleteWorkItem` and `HandleBlockingFiber` functions. The former simply completes the given work item and reschedules all effects that are waiting on this work item as per the design.

This, however, causes an issue. Consider the scenario when an effect that is blocked by a fiber is about to be rescheduled to the blocking worker on line 17. It may be the case, that before the work item is added to the fiber it is waiting on, that the fiber is completed in the meantime. This means – since the work item was not added to the list of blocked effects – that it will never get to retrieve data and thus a deadlock is present. This is solved by using the auxiliary function `HandleBlockingFiber` on line 18 right after the fiber has been rescheduled. It simply checks if the fiber is already complete after being

added, and thus reschedules any work items that may have been leftover.

```
type internal EvalWorker(runtime: Runtime,
  workItemQueue: BlockingCollection<WorkItem>,
  blockingWorker: BlockingWorker, evalSteps) as self =

  let _ = (async {
    for workItem in workItemQueue.GetConsumingEnumerable() do
      match runtime.LowLevelRun workItem.Eff workItem.PrevAction
      ↪  evalSteps workItem.Stack with
      | (Success res, _), Evaluated, _ ->
          self.CompleteWorkItem workItem (Ok res)
      | (Failure err, _), Evaluated, _ ->
          self.CompleteWorkItem workItem (Error err)
      | (eff, stack), RescheduleForRunning, _ ->
          let workItem = WorkItem.Create eff stack
          ↪  workItem.LLFiber RescheduleForRunning
          workItemQueue.Add workItem
      | (eff, stack), RescheduleForBlocking blockingItem, _ ->
          let workItem = WorkItem.Create eff stack
          ↪  workItem.LLFiber (RescheduleForBlocking
          ↪  blockingItem)
          blockingWorker.RescheduleForBlocking blockingItem
          ↪  workItem
          self.HandleBlockingFiber blockingItem
      | _ -> failwith $"EvalWorker: Error occurred while
      ↪  evaluating effect!"
  } |> Async.StartAsTask |> ignore)

member private _.CompleteWorkItem workItem res =
  workItem.Complete res
  workItem.LLFiber.RescheduleBlockingWorkItems workItemQueue

member private _.HandleBlockingFiber blockingItem =
  match blockingItem with
  | BlockingFiber llfiber ->
      if llfiber.Completed() then
        llfiber.RescheduleBlockingWorkItems workItemQueue
  | _ -> ()
```

Listing 42: Implementation of the advanced interpreter's evaluation worker with removed deadlock detector compilation symbols

5.2.3.2 Blocking worker

The blocking worker is the element with the largest amount of changes when compared to the intermediate blocking worker. The blocking worker implementation of the advanced interpreter is found in Listing 43. As per the design, the blocking worker now only handles channels. On line 6, the blocking worker iterates through incoming events which is the channel that just retrieved data as mentioned previously. The blocking worker then simply calls the `RescheduleBlockingWorkItem` function of the channel and passes the work queue. As seen in Section 5.1.3 (Listing 31) this function takes one waiting effect – if there is one – and puts it back into the work item queue for interpretation. The `RescheduleForBlocking` function was previously seen used by the interpreter.

```
1   type internal BlockingWorker(
2     workItemQueue: BlockingCollection<WorkItem>,
3     blockingEventQueue: BlockingCollection<Channel<obj>>) =
4
5     let _ = (async {
6       for blockingChan in
          ↪ blockingEventQueue.GetConsumingEnumerable() do
7         if blockingChan.HasBlockingWorkItems() then
8           blockingChan.RescheduleBlockingWorkItem workItemQueue
9         else
10          blockingEventQueue.Add blockingChan
11    } |> Async.StartAsTask |> ignore)
12
13  member internal _.RescheduleForBlocking blockingItem workItem =
14    match blockingItem with
15    | BlockingChannel chan ->
16        chan.AddBlockingWorkItem workItem
17    | BlockingFiber llfiber ->
18        llfiber.AddBlockingWorkItem workItem
```

Listing 43: Implementation of the advanced interpreter's blocking worker with removed deadlock detector compilation symbols

5.2.4 Debugging tools

In this section, the two debugging tools presented in Section 4.2.6 will have their implementation and usage described. It should be noted, that both the data

structure monitor and the deadlock detector only support the intermediate and advanced interpreters.

5.2.4.1 Data structure monitor

The implementation of the data structure monitor can be found in Listing 44. The monitor is implemented as a type that takes critical data structures such as the `workItemQueue` through its constructor. On line 3 and 4, notice that the accepted data structures are wrapped by the `Option` type. This is simply to support both the intermediate and advanced interpreters, as the `BlockingItemQueue` is only used by the intermediate and `BlockingEventQueue` by the advanced.

On line 6 through 16, an endless loop is started in a separate OS thread that simply prints out the contents of the data structures every second. This is done by a function corresponding to each data structure, for example, for the `workItemQueue`, it is the `PrintWorkItemQueueInfo` function that has the task of printing the information to the console. The implementation of the functions `PrintBlockingItemQueueInfo` and `PrintBlockingEventQueueInfo` is not shown as it is similar to that of `PrintWorkItemQueueInfo` albeit printing other elements. Taking a closer look at `PrintWorkItemQueueInfo`, it simply iterates through – without removing – all items in the work item queue and prints out characteristics such as:

- Is the work item's fiber completed?
- How many blocking effects are waiting on the work item's fiber?
- What is the previous action of the work item?
- What is the effect of the work item?

The `Monitor` implementation is guarded by a compilation directive called `MONITOR`. In other words, the `MONITOR` flag will have to be added as an compilation option to the compiler if the data structure monitor is wanted to be used. In a similar manner, the usage of the type is guarded as well. For example, consider the usage of the monitor for the advanced interpreter as shown in Listing 45. The code inside the `if` and `endif` directives will only be compiled if the `MONITOR` flag is present at compilation. To showcase an application of the monitor, the program from Listing 7 will be executed with data structure monitoring support by the advanced interpreter. Consider the output from the execution seen in Listing 46. The monitor provides the information that 1 work item is present in the `workItemQueue` and that it is already completed. In addition, several blocking events are present in the `BlockingEventQueue`. This makes sense, as this specific program sends a lot of messages.

```
type internal Monitor(
  workItemQueue: BlockingCollection<WorkItem>,
  blockingItemQueue: Option<BlockingCollection<BlockingItem *
    WorkItem>>,
  blockingEventQueue: Option<BlockingCollection<Channel<obj>>>)
    as self =

  let _ = (async {
    while true do
      self.PrintWorkItemQueueInfo workItemQueue
      match blockingItemQueue with
      | Some queue -> self.PrintBlockingItemQueueInfo queue
      | _ -> ()
      match blockingEventQueue with
      | Some queue -> self.PrintBlockingEventQueueInfo queue
      | _ -> ()
      System.Threading.Thread.Sleep(1000)
  } |> Async.StartAsTask |> ignore)

  member private _.PrintWorkItemQueueInfo (queue :
    BlockingCollection<WorkItem>) =
    printfn $"MONITOR: workItemQueue count: %i{queue.Count}"
    printfn "MONITOR: ------------ workItemQueue information
      start ------------"
    for workItem in queue.ToArray() do
      let llfiber = workItem.LLFiber
      printfn $"MONITOR:    ------------ workItem start
        ------------"
      printfn $"MONITOR:      WorkItem LLFiber completed:
        %A{llfiber.Completed()}"
      printfn $"MONITOR:      WorkItem LLFiber blocking items
        count: %A{llfiber.BlockingWorkItemsCount()}"
      printfn $"MONITOR:      WorkItem PrevAction:
        %A{workItem.PrevAction}"
      printfn $"MONITOR:      WorkItem Eff: %A{workItem.Eff}"
      printfn $"MONITOR:    ------------ workItem end
        ------------"
    printfn "MONITOR: ------------ workItemQueue information end
      ------------"

  member private _.PrintBlockingItemQueueInfo (queue :
    BlockingCollection<BlockingItem * WorkItem>) =
    // logic <snip>

  member private _.PrintBlockingEventQueueInfo (queue :
    BlockingCollection<Channel<obj>>) = // logic <snip>
```

Listing 44: Data structure monitor implementation

```
1  #if MONITOR
2  Utils.Monitor(workItemQueue, None, Some blockingEventQueue,
   ↪  Some <| blockingWorkItemMap.Get())
3  |> ignore
4  #endif
```

Listing 45: Data structure monitor compilation guard for the advanced interpreter

```
1  MONITOR: workItemQueue count: 1
2  MONITOR: ------- workItemQueue information start -------
3  MONITOR:    ------------ workItem start ------------
4  MONITOR:      WorkItem LLFiber completed: true
5  MONITOR:      WorkItem LLFiber blocking items count: 0
6  MONITOR:      WorkItem PrevAction: Evaluated
7  MONITOR:      WorkItem Eff: SequenceSuccess
8    (SendMessage (42, FSharp.FIO.FIO+Channel`1[System.Object]),
9     <fun:UpcastError@154-1>)
10 MONITOR:    ------------ workItem end ------------
11 MONITOR: ------- workItemQueue information end -------
12
13
14 MONITOR: blockingEventQueue count: 69575
15 MONITOR: ------- blockingEventQueue information start -------
16 MONITOR:    ------------ blockingChan start ------------
17 MONITOR:      Count: 69576
18 MONITOR:    ------------ blockingChan end ------------
19 MONITOR:    ------------ blockingChan start ------------
20 MONITOR:      Count: 70220
21 MONITOR:    ------------ blockingChan end ------------
22 ...
```

Listing 46: Data structure monitor execution example of Listing 7

5.2.4.2 Deadlock detector

The implementation of the deadlock detector is presented in Listing 47. The detector is implemented as a type called `DeadlockDetector` that is generic over the type of used evaluation workers and blocking worker. This is necessary such that the detector is able to support both the intermediate and advanced runtimes. The `DeadlockDetector` accepts the `workItemQueue` and an interval

in milliseconds called `internalMs`, which is the rate for the detector to check whether a deadlock is present.

Whenever a blocking item is met by the interpreter, it is saved by the detector in the `blockingItems` dictionary. This way, the detector has total knowledge of all effects that are blocking and whether any effects are waiting to be interpreted in the `workItemQueue`. In addition, it also has access to references for all evaluation workers and blocking workers.

Examining the implementation, on line 9 a new OS thread is spawned that starts an endless loop. This loops checks if there are no work items left, if all the evaluation workers are idle, and if there is blocking items present. In the above situation, because there are no work items in the queue ready to be interpreted, and none of the evaluation workers are currently working, the blocking items that are waiting will never receive data and thus a deadlock is present.

However, with the current implementation of the detector, some false positives of whether all the workers are idle may take place. To combat this, a simple countdown system has been implemented in the detector. The detector will have to detect the scenario 10 times with `internalMs` milliseconds between each occurrence, and only then will the detector report that a deadlock is present. This is simply just to increase the confidence of a deadlock being present.

```
type internal DeadlockDetector<'B, 'E when 'B :> Worker and 'E :> Worker>(
  workItemQueue: BlockingCollection<WorkItem>,
  intervalMs: int) as self =
  let blockingItems = new ConcurrentDictionary<BlockingItem, Unit>()
  let mutable blockingWorkers : List<'B> = []
  let mutable evalWorkers : List<'E> = []
  let mutable countDown = 10

  let _ = (async {
    while true do
      if workItemQueue.Count <= 0
         && self.AllEvalWorkersIdle()
         && blockingItems.Count > 0
      then
        if countDown <= 0 then
          printfn "DEADLOCK_DETECTOR: ############ WARNING: Potential deadlock detected! ############"
          printfn "DEADLOCK_DETECTOR:     Suspicion: No work items left, All EvalWorkers idling, Existing blocking items"
        else
          countDown <- countDown - 1
      else
        countDown <- 10
      System.Threading.Thread.Sleep(intervalMs)
  } |> Async.StartAsTask |> ignore)

  member internal _.AddBlockingItem blockingItem =
    blockingItems.TryAdd (blockingItem, ())
    |> ignore

  member internal _.RemoveBlockingItem (blockingItem: BlockingItem) =
    blockingItems.TryRemove blockingItem |> ignore

  member private _.AllEvalWorkersIdle() =
    not (List.contains true <|
      List.map (fun (evalWorker: 'E) ->
        evalWorker.Working()) evalWorkers)

  member private _.AllBlockingWorkersIdle() =
    not (List.contains true <|
      List.map (fun (evalWorker: 'B) ->
        evalWorker.Working()) blockingWorkers)
```

Listing 47: Deadlock detector implementation

5.3 Summary

In this chapter, the implementation of FIO's effect and runtime systems were shown and explained. First of all, for the effect system, the implementation of the spawn, await, >>, ||| and |||* functions from the effect API were presented and discussed. It was shown that a lot of explicit type casting and other inconveniences are hidden away by the API functions.

Moving along, the final implementation of the interpreter structure was presented, using the ADT with type casting approach. Implementation details such as how the opcodes are ensured to only be accessible internally was shown together with upcasting functions. Next, the final implementation of the channel type was shown. It was emphasized on how thread-safety was ensured by using the BlockingCollection<obj> type as its internal data structure. Other technical details such as how blocking effects are kept in a queue for the advanced interpreter as well as the idea behind its DataCounter to solve a race-condition. The final implementation of both the LowLevelFiber and Fiber types were presented with emphasis on how the 3 rules of **R1**, **R2** and **R3** are satisfied by the implementation.

For the runtime system, it was mentioned that all interpreters are tail-call optimized, meaning that recursive calls do not allocate stack frames. The naive interpreter is presented with 3 auxiliary functions that maintain the virtual stack frame of the interpreter. It was explained that the virtual stack frame is used be able to apply the correct continuations made by the SequenceSuccess and SequenceError opcodes. Next, the two interpretation functions of LowLevelRun and Run was presented showing how the logic for each opcode is implemented.

Furthermore, the intermediate interpreter implementation is presented with its 3 auxiliary functions. The work item type that is used by the evaluation and blocking workers was introduced as well, followed by the interpretation functions. It is shown how the intermediate interpreter reschedules out blocking effects to the blocking worker and how spawning a concurrent effect is simply just adding a new work item to the queue of the evaluation workers, thereby realizing the concept of green threads. Next, the evaluation worker was presented together with the Action type consisting of the Evaluated, RescheduleForRunning and RescheduleForBlocking cases. The evaluation worker reacts differently upon which of the actions is returned from the LowLevelRun function. Moreover, the blocking worker is presented and how it realizes its linear check through the blocked effects was explained. A race-condition in regards blocking effects that await on the same opcode was introduced and the solution explained.

Moving on to the advanced interpreter, it was shown that the `LowLevelRun` function is similar to the one of the intermediate however with the slight change that the `SendMessage` opcode now sends an event to the blocking worker as well. The evaluation worker was then presented with its new functionality on how it turns the rescheduling of blocked effects for fibers constant. In addition, an issue of adding work items to a fiber that is completed before the blocking work item is added was presented and solved. At last, the blocking worker's implementation was presented and shown how it processes events sent from the `SendMessage` opcode. Finally, the implementations of the data structure monitor and the deadlock detector was presented and explained.

In the upcoming evaluation chapter, the implementation of FIO will be evaluated by benchmarking the execution time and scalability of a variety of concurrent programming aspects.

Chapter 6

Evaluation

In this chapter, the evaluation methodology for FIO is presented. The methodology is the implementation of 5 benchmarks using FIO which together will determine if and how much the design of the intermediate and advanced interpreters has improved upon the naive interpreter. The benchmark suite consists of the *Pingpong*, *Threadring*, *Big*, *Bang* and *Spawn* benchmarks that each measure the performance of an aspect of concurrent computing. They will be run by the 3 interpreters with different parameters and will be evaluated on their performance in terms of execution time and scalability. First, the benchmarks will be presented briefly together with the hardware they are ran on. Then, the high precision timing technique used for timing the benchmarks is presented together with an explanation of its accuracy. Furthermore, each benchmark will be described in regards to how it works and what it measures. Moreover, the results will be examined and a conclusion on whether the expected results were obtained will be presented. Finally, a summary of the chapter is given.

6.1 Methodology

The chosen evaluation methodology for FIO is benchmarking. The interpreters will be evaluated by implementing 5 different benchmarks that each have been chosen on the basis of measuring a key metric of concurrent computing. Each benchmark will be evaluated on how well it performs in terms of execution time and how well it scales as the amount of fibers increase.

2 benchmarks are used from [IS14], which is a paper providing a standard benchmark suite for comparing the performance of actor applications. More specifically, the *Pingpong* and *Threadring* benchmarks are used from this paper. These benchmarks will be able to determine how well an interpreter performs when it comes to messaging and context-switching between fibers.

An additional 2 benchmarks are chosen from [APR$^+$12], which is a paper that provides scalability benchmarks for Erlang. These benchmarks will help determine how well the interpreters scale when it comes to concurrency. The chosen benchmarks from this paper are called *Big* and *Bang*, and they will measure how well message passing with a large amount of fibers is handled. The fifth benchmark is called *Spawn* and measures spawning time of fibers.

The performance of a benchmark will be measured in how long it takes for an interpreter to complete interpretation, also called execution time. This time will be measured in milliseconds with a high precision timing technique which will be discussed in detail in Section 6.3. Before each benchmark is discussed in detail, a small overview can be found in Table 6.1.

Benchmark	Metric	Origin
Pingpong	Message sending and retrieval	[IS14]
Threadring	Message sending and retrieval, context switching between fibers	[IS14]
Big	Contention on channel, many-to-many message passing	[APR$^+$12]
Bang	Many-to-one message passing	[APR$^+$12]
Spawn	Spawning time of fibers	–

Table 6.1: Overview of benchmarks that will be used to evaluate the performance of FIO

6.2 Hardware specifications

The benchmarks will be run on a Lenovo ThinkPad X1 Carbon 7th generation device with specifications as shown in Table 6.2. The device will be plugged in to a power source when the benchmarks are run to provide stable performance. This may be relevant in the case of result replication or comparing the benchmarks on devices with different hardware configurations.

Being aware of the amount of physical cores and supported threads of the machine that the benchmarks are run on is critical to determine the optimal number of threads used per CPU core. It is usually recommended to use 1 or 2 threads per core for optimal efficiency.

Component	Hardware
CPU	Intel(R) Core(TM) i7-8665U 1.90GHz with 4 physical cores and 8 threads
RAM	16GB DDR4 2.133MHz

Table 6.2: Relevant hardware specifications of the Lenovo ThinkPad X1 Carbon (7th gen.) that the benchmarks are ran on

6.3 High precision timing

To make sure the timing of the benchmarks is being measured as accurately as possible, they need to implement a high precision timing technique. This is achieved by using a high resolution stopwatch available in the .NET framework. [Cora]

The `stopwatch` class provides a set of methods and properties that can be used to accurately measure elapsed time. It measures elapsed time by counting timer ticks in the underlying timer mechanism. The underlying timer mechanism can either be the system timer or a high resolution performance counter if the hardware supports it. In the case of the X1 Carbon used in this thesis, the `stopwatch` implementation uses a high resolution performance counter.

In more technical terms, the accuracy of the `stopwatch` depends on the underlying hardware. For the X1 Carbon, a timer frequency of 10.000.000 ticks per second is reported by the .NET framework. This is equivalent to a timer resolution of one tick per 100 nanoseconds, which is the smallest unit of time that can be measured accurately by the timer. For this reason, it is important that

all benchmarks measure at least 100 nanoseconds, otherwise the measurements may not be accurate. [Corb]

6.4 Benchmark suite

The benchmark suite consists of the 5 benchmarks shown previously in Table 6.1. Each benchmark is parameterized by the amount of fibers it spawns and the amount of rounds. This is however not true for *Pingpong* and *Spawn*, as *Pingpong* will always spawn 2 fibers and *Spawn* does not use rounds. The exact meaning of a round changes from benchmark to benchmark and will therefore be explained further in upcoming sections.

The parameters used for the benchmarks can be seen in Table 6.3. The number of rounds are low as the focus is on scalability of the interpreters concurrency implementations. As the number of spawned fibers is constant for *Pingpong*, a large number of rounds is used. For the *Threadring*, *Bang* and *Spawn* benchmarks, a number of 5.000 fibers was chosen as a seemingly realistic amount. *Big* only uses 500 fibers as it is a computationally expensive benchmark and would otherwise take too long to interpret.

Benchmark	Spawned fibers	Rounds
Pingpong	2 (constant)	120.000
Threadring	5.000	1
Big	500	1
Bang	5.000	1
Spawn	5.000	–

Table 6.3: The set of parameters that each benchmark is ran with

6.4.1 Pingpong

The first benchmark of the suite is the *Pingpong* benchmark. It is nearly identical to the example shown previously in Chapter 2 (Listing 6), however, to avoid any source of confusion it will still be explained.

The *Pingpong* benchmark spawns 2 fibers called *pinger* and *ponger*. These fibers exchange messages between each other, back and forth, for 120.000 rounds. A single round consists of *pinger* sending a “ping” message to *ponger* that is

awaiting the message. Once received by *ponger*, *ponger* replies back with a "pong" message to *pinger*. The messages are integer values.

The benchmark measures the performance of message sending and retrieval by repeatedly sending and receiving messages. Since 2 messages are sent and received per round, it is a total of 220.000 sent and 220.000 received messages that is tested in this benchmark. This will help determine the execution speed of messaging.

The benchmark solely measures the execution time. This is achieved by letting *pinger* handle when the stopwatch is started and stopped. When both fibers are spawned, *ponger* sends a "ready" message to *pinger* to let it know that it is spawned and ready. Once *pinger* has received that message, it starts the stopwatch and immediately starts sending messages. Once *pinger* has received the last reply from *ponger*, the stopwatch is stopped and the elapsed time is recorded as the measured execution time of the benchmark.

6.4.2 Threadring

The second benchmark is that of *Threadring*. The *Threadring* benchmark spawns 5.000 fibers that are connected in a ring, where a token (a message) is passed along the ring for 1 round.

For example, a *Threadring* benchmark with 3 spawned fibers and 1 round would look like the following. 3 fibers are spawned, *f1*, *f2* and *f3*. All fibers start out by awaiting the token, so first the token is injected into *f1* to start the benchmark. Once received, *f1* passes the token to *f2*, which in turn passes it to *f3*, which in turn passes it to *f1* and the benchmark is complete. If more rounds were present, the token would go around the ring for the number of rounds. The passed token is an integer value.

The benchmark measures the performance of message sending and retrieval together with context switching between fibers. As the token is passed along the ring, there is a repeated switch from fiber to fiber. This means that only one fiber is active at any time, either sending or receiving a message.

Threadring measures execution time. This is made sure by having an additional fiber that handles the timing by awaiting a "ready" message from each of the 5.000 spawned fibers. Once 5.000 "ready" messages are received, it is known that all the fibers are spawned and ready. The stopwatch is then started and the token is injected into a fiber in the ring to start the messaging. Once a fiber has received and sent the token equal to the amount of rounds, a "stop"

message is sent to the timing fiber. Once 5.000 “stop” messages are received the stopwatch is stopped and the measurement of the benchmark is done.

6.4.3 Big

Big is a benchmark that spawns 500 fibers where each fiber sends messages to every other fiber for 1 round. A round consists of each fiber sending a “ping” message to every other fiber. A fiber responds back with a “pong” message to any “ping” message it receives.

The benchmark measures the performance of many-to-many messaging as each fiber is sending messages to every other fiber. Each fiber may concurrently be receiving messages from other fibers as well, meaning that the benchmark measures the effects of contention on the fibers’ channel as well. Each message sent by the fibers are integer values.

The execution time is measured by the spawned fibers sending a “ready” message to a timing fiber to indicate that they are spawned and ready. Once the timing fiber has received 500 “ready” messages, one from each fiber, the stopwatch is started and the timing fiber sends a “go” message to each of the fibers. Once a fiber has received a “go” message, it starts sending “ping” messages. Once a fiber has received 499 “pong” messages, it sends a “stop” message to the timing fiber. Once the timing fiber has received 500 stop messages, it stops the stopwatch and the benchmark is complete.

6.4.4 Bang

The next benchmark in the suite is *Bang*. *Bang* is a benchmark that spawns 5.000 fibers that all send 1 message to a single receiving fiber for 1 round. In this benchmark, a round is equal to the amount of messages sent to the receiving fiber. It measures the performance of many-to-one messaging as the 5.000 fibers flood the single receiver with messages of integer values.

The benchmark measures execution time by having the 5.000 spawned fibers plus the receiving fiber send a “ready” message to a timing fiber. Once the timing fiber has received 5.001 “ready” messages, it sends a “go” message to the sending fibers to indicate they can start sending and to the receiving fiber to indicate that it should now start receiving. Once the receiving fiber has received 5.000 messages, it sends a “stop” message to the timing fiber to stop the stopwatch and the benchmark terminates.

6.4.5 Spawn

The *Spawn* benchmark does not measure any kind of message passing but the time it takes to spawn fibers. It simply starts the stopwatch, spawns 5.000 fibers that each send a “stop” message to a timing fiber once they are spawned. Once 5.000 “stop” messages have been received, it is known that all the fibers have been spawned and the timing fiber stops the stopwatch to complete the benchmark.

6.5 Results

Each benchmark from the suite will be evaluated in 9 different scenarios as shown in Table 6.4. The benchmarks will each be interpreted 30 times in each scenario, such that 30 time measurement samples for each scenario is used for the results. Since the CPU of the X1 Carbon has support for 8 threads, the advanced and intermediate interpreters will be used with both 8 and 16 workers, using 1 and 2 threads per core respectively. More precisely, the interpreters will use configurations of 7 and 15 evaluation workers with a single blocking worker. Both of these configurations will be run with 15 and 100 evaluation steps.

Due to the amount of fibers and rounds used in each of the benchmarks, reaching a minimum measurement for each interpretation of 100 nanoseconds will not be a problem. In addition, it should be noted that the 30 measurements are executed by the same instance of the interpreter and that the benchmarks are ran in release mode for performance optimized binaries.

Scenario	Interpreter	Evaluation workers	Evaluation steps
1	Naive	–	–
2	Intermediate	7	15
3	Intermediate	7	100
4	Intermediate	15	15
5	Intermediate	15	100
6	Advanced	7	15
7	Advanced	7	100
8	Advanced	15	15
9	Advanced	15	100

Table 6.4: The 9 scenarios that each of the benchmarks from Table 6.1 will go through using the parameters given in Table 6.3

6.5.1 Pingpong

The results for the *Pingpong* benchmark can be seen in Figure 6.1. The results of each scenario from Table 6.4 has been plotted as a boxplot as seen on the x-axis. `I` stands for the intermediate interpreter and `A` for the advanced interpreter. `EW` and `ES` stands for the number of evaluation workers and evaluation steps respectively. For example, Naive represents the first scenario from the table and I (EW: 7 ES: 15) the second. The y-axis represents the amount of time it has taken for a given interpreter to execute the benchmark, meaning that a lower value equals better performance. For example, it is observed that the median execution time for the advanced interpreter with 7 evaluation workers and 15 evaluation steps is around 500 milliseconds (ms).

Taking a closer look at Figure 6.1, it is noticeable that nothing interesting happens, however with good reason. The *Pingpong* benchmark only spawns 2 fibers, which means that it is not going to take any advantage of the scheduling improvements introduced with the intermediate and advanced interpreters. In fact, this can be seen directly, as the execution time increases with the intermediate and advanced interpreters due to additional overhead. The reason why the advanced interpreter performs worse than the intermediate is due to the added overhead of sending an event every time a message is sent. The *Pingpong* benchmark does send a large amount of messages, so it would be expected for this change to have a small impact on performance as observed.

It is expected for this benchmark for the naive interpreter to be the best performing, which is the case. Only 2 OS threads are spawned, so no scaling issues are present. However, because it is OS threads, some spread is expected for the naive interpreter as observed. This is because the threads are more susceptible to disturbance from other running processes and interruptions from the CPU than fibers. An additional observation is that the scenarios using 1 thread per core seems to perform slightly better than using 2. In addition, since both fibers have to continuously send and receive messages, they must both stay alive for the whole duration of the benchmark.

Overall, these results show that the improvements made to the intermediate and advanced interpreters only decreased the performance of message sending. This is expected as for message sending, only overhead has been added. No scalability plot is created for *Pingpong* as the number of fibers is constant.

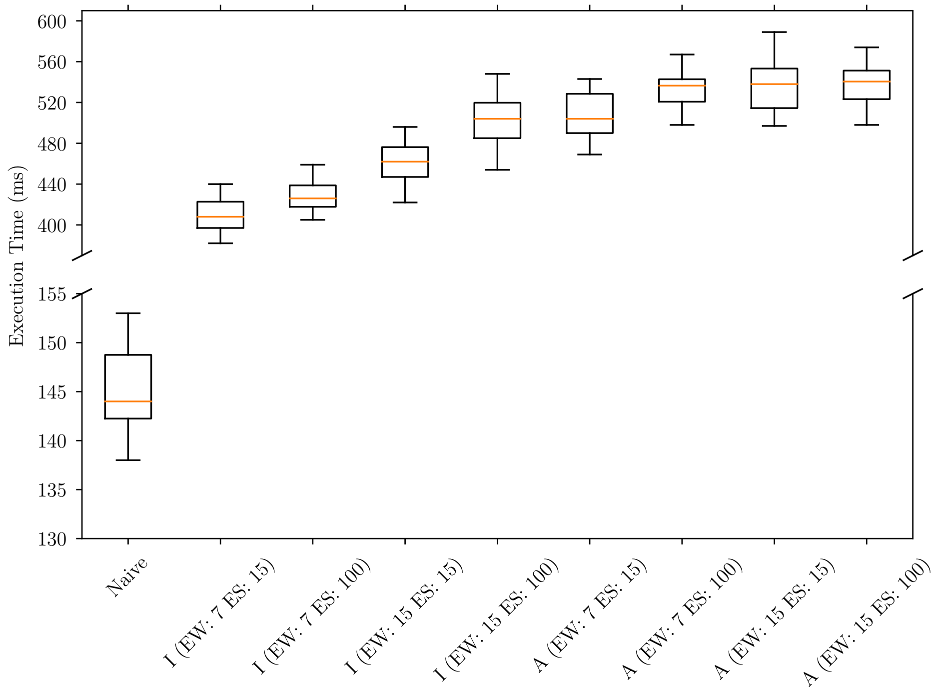


Figure 6.1: Boxplot of the *Pingpong* benchmark results with 2 fibers and 120.000 rounds

6.5.2 Threadring

The benchmark results for the *Threadring* benchmark is shown in Figure 6.2. Examining the figure, it is clear that the intermediate interpreter performs a lot worse than the naive, meanwhile the advanced interpreter is the best performing.

These results are expected as *Threadring* is a benchmark that is sensitive to the rescheduling of blocked fibers being delayed. The issue with the intermediate interpreter is that it checks the blocking fibers in a linear manner as explained in Section 4.2.5. Worst case the blocking worker will have to check all items in its queue before it reaches a fiber that is not blocking anymore. Due to the fibers being dependent on each other and that only one fiber is active at a time, evaluation workers may have to wait for a significant amount of time before fibers are rescheduled for interpretation. This will massively decreases performance as observed.

However, introducing the improvement of acting upon blocked fibers immediately in the advanced interpreter increased the performance to be not only better than the intermediate, but also the naive. The reason for the significant boost in performance is that the previous issue of having to wait on a fiber that already has received data, yet is still in the blocking queue is not present anymore. It can also be observed, that the advanced interpreter has less outliers and spread than the intermediate, which is most likely due to this change as well. The lifetime of fibers in *Threadring* is different to *Pingpong* because fibers are allowed to die after they have received and sent their messages. This may mean that not all 5.000 fibers are active at the same time except for at the very start of the benchmark, which could affect performance positively.

This shows that the advanced interpreter handles scenarios with a large amounts of blocking, context-switching and low amount of messages very efficiently as expected due to the accelerated handling of blocked fibers.

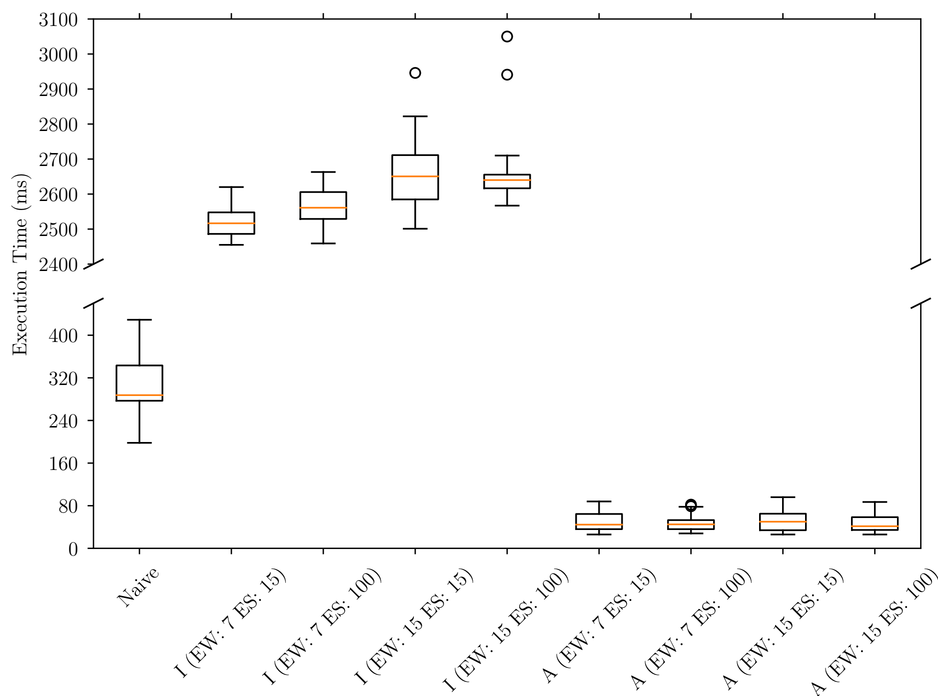


Figure 6.2: Boxplot of the *Threadring* benchmark results with 5.000 fibers and 1 round

A comparison between the scalability of the naive, intermediate and advanced interpreters using 7 evaluation workers and 15 steps is presented in Figure 6.3. The x-axis shows the amount of fibers and the y-axis the mean execution time of 30 measurements. As expected with the results given in Figure 6.2, the intermediate interpreter scales the worst and the advanced the best with the naive in-between. The *O3* objective has been satisfied by the advanced interpreter.

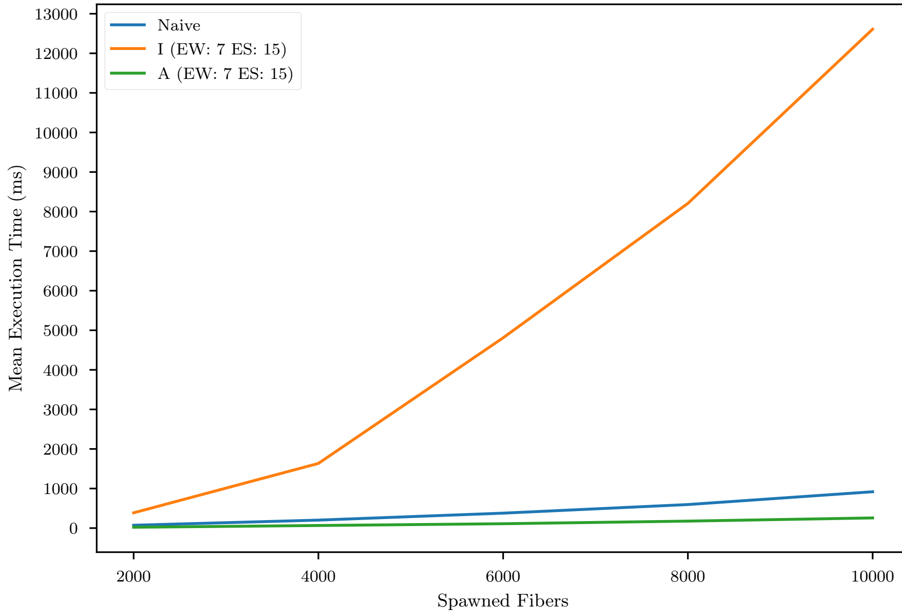


Figure 6.3: Scalability plot of the *Threadring* benchmark showing how the amount of fibers relates to execution time

6.5.3 Big

The next benchmark is *Big* with its results presented in Figure 6.4. The intermediate interpreter is the best performing and the advanced the worst. The naive interpreter is in-between with slightly more spread and significant outliers.

The results are slightly surprising as it was not expected for the advanced interpreter to perform worse than the intermediate. It is suspected that the reason is the chaotic nature of many-to-many messaging as there is a lot of message sending between all of the fibers. The improved handling of blocked fibers in the advanced interpreter may not be contributing significantly to increase the

performance as handling blocked fibers is not the primary action in this benchmark. For the intermediate interpreter, the queue of blocked fibers will contain multiple fibers that are ready to be rescheduled back for interpretation, so the linear checking is less of a performance bottleneck compared to *Threadring*.

The reason the advanced interpreter performs worse than the intermediate can only be due to the additional overhead of sending an event to the blocking worker every time a message is sent. As a lot of messages are sent in this particular benchmark, it would be expected for this to be noticeable in the performance. In addition, the evaluation worker used for the advanced interpreter checks whether a fiber has any blocked effects waiting when its completed which also adds additional overhead. Furthermore, the naive interpreter has larger spread compared to the other interpreters and significant outliers. This is again due to the OS threads being prone to disturbances which will be especially visible in a computationally heavy benchmark like *Big*. Due to the immense amount of messages, the fibers will have to stay alive for the whole duration of the benchmark. For an overall conclusion for *Big*, it is observed that the intermediate interpreter handles many-to-many messaging with moderate amounts of blocking the best.

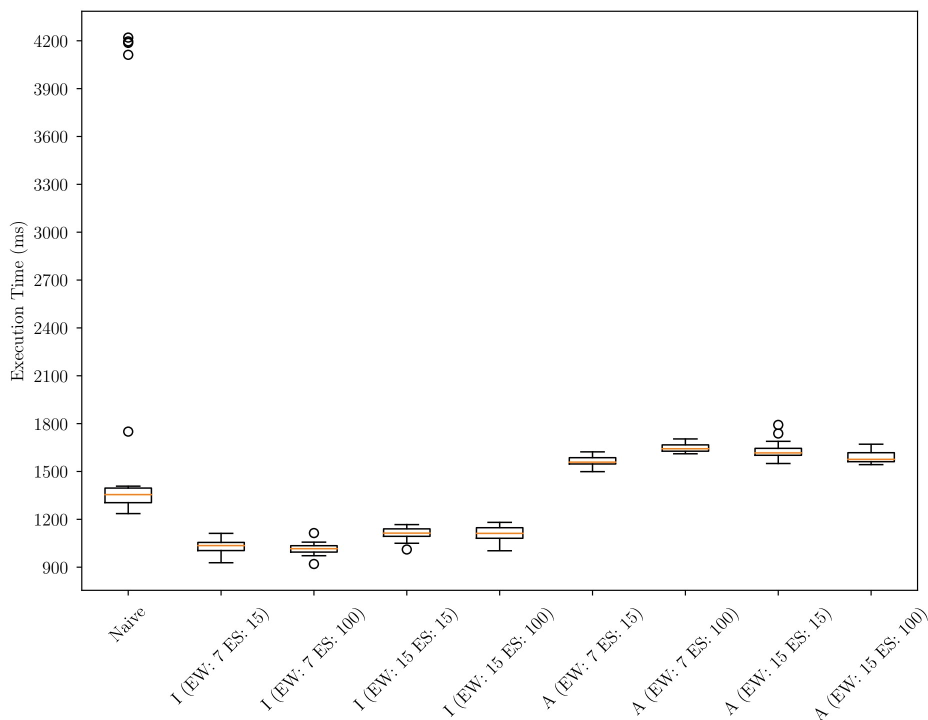


Figure 6.4: Boxplot of the *Big* benchmark results with 500 fibers and 1 round

The scalability comparison of the *Big* benchmark is found in Figure 6.5. In Figure 6.4 it was observed that the advanced interpreter performed worse than the naive. An interesting observation here is that the advanced interpreter scales better than the naive as the amount of fibers increase, meaning that if more than 500 fibers were used for the previous results, the advanced interpreter would perform better than the naive. For *Big*, both the intermediate and advanced interpreters satisfies the *O3* objective.

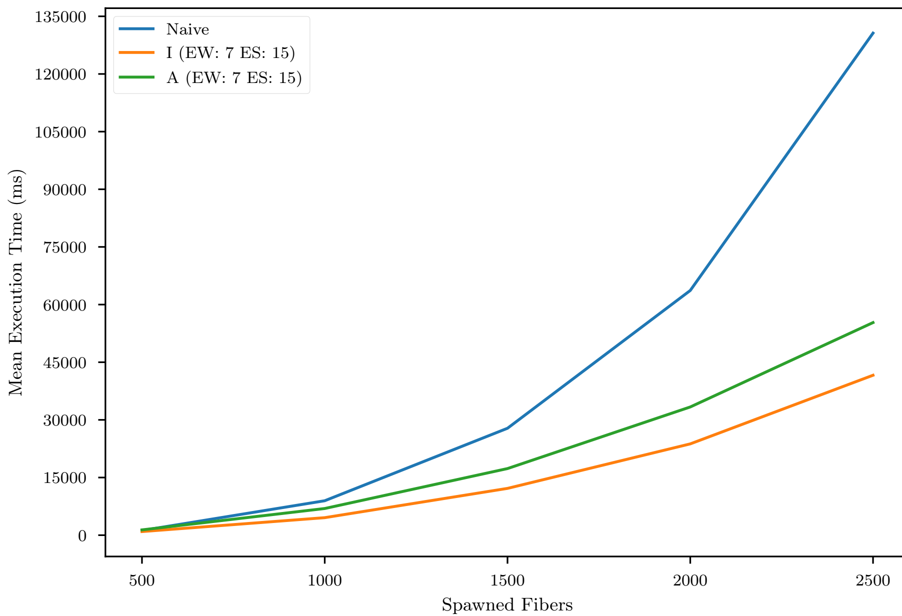


Figure 6.5: Scalability plot of the *Big* benchmark showing how the amount of fibers relates to execution time

6.5.4 Bang

The results for the *Bang* benchmark can be seen in Figure 6.6. The naive is the worst performing interpreter meanwhile the intermediate and advanced are close to being equal however with the latter containing significantly more spread.

One interesting observation is that *Bang* – like the *Pingpong* benchmark – solely focuses on messaging, yet the naive interpreter is the best performing in *Pingpong* and the worst in *Bang*. One difference, however, is the lifetime of the fibers. For *Pingpong* the fibers stay alive for the whole duration of the bench-

mark which is not the case for *Bang*. When a fiber in the *Bang* benchmark has sent its message, it is free to die as no more work is left for it to do. This may decrease the amount of fibers alive at the same time, and since fibers in the intermediate and advanced interpreters are more light-weight than threads in the naive, better performance is achieved.

Another obvious observation is the significant spread of the advanced interpreter when compared to the intermediate. This is most probable due to the extra step of sending an event whenever a message is sent through a channel.

Overall, it is observed that the adjustments to the interpreters has greatly improved performance when it comes to many-to-one messaging with no blocking. More precisely, it is likely not due to the changes of handling fibers, but due to using green threads rather than OS threads.

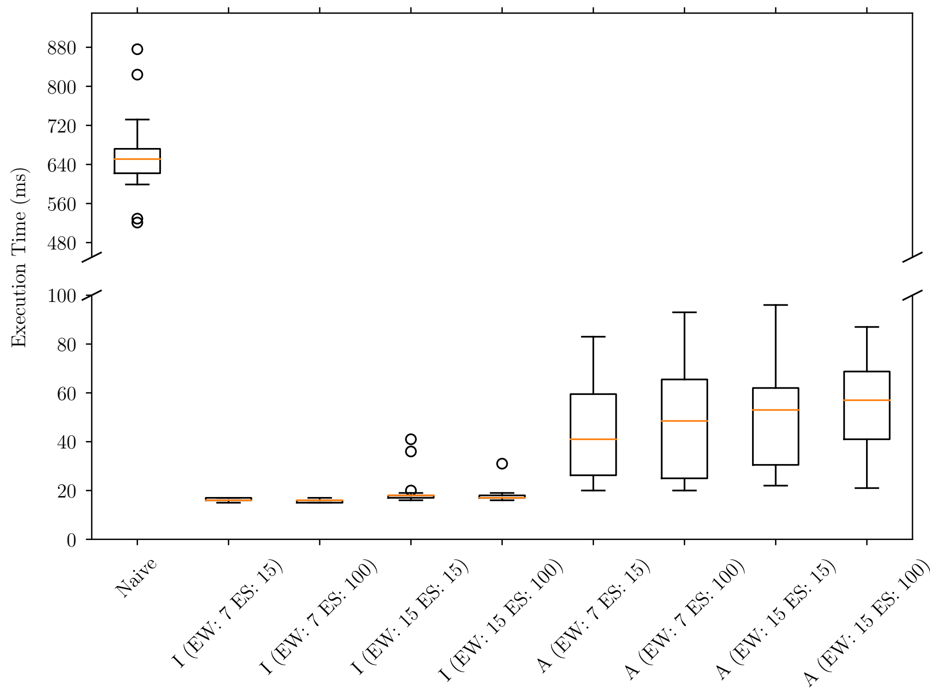


Figure 6.6: Boxplot of the *Bang* benchmark results with 5.000 fibers and 1 round

Once more, a scalability plot can be found in Figure 6.7. It was expected that the intermediate and advanced interpreters would scale better than the naive,

however not as well as observed. The reason for linear scaling may be explained by the lifetime of the fibers and thus *O3* is satisfied both by the intermediate and advanced interpreters.

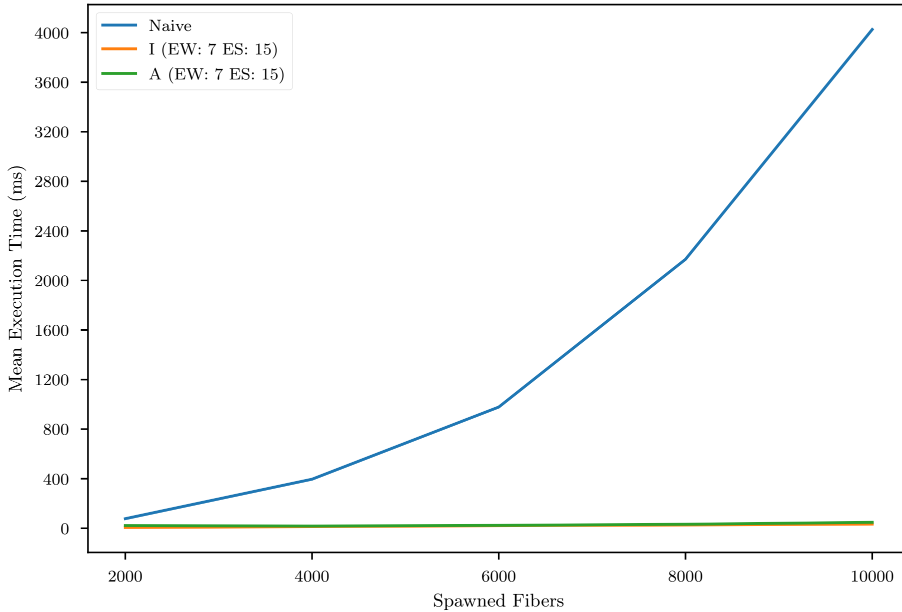


Figure 6.7: Scalability plot of the *Bang* benchmark showing how the amount of fibers relates to execution time

6.5.5 Spawn

The results for the *Spawn* benchmark is presented in Figure 6.8. Examining the figure, it can be observed that the naive interpreter is performing worse compared to the intermediate and advanced interpreters, with the advanced performing slightly worse than the intermediate.

The reason why the intermediate and advanced interpreters are faster than the naive is because it costs less resources to spawn a fiber than a thread. It is not obvious as to why the advanced interpreter is slightly slower than the intermediate.

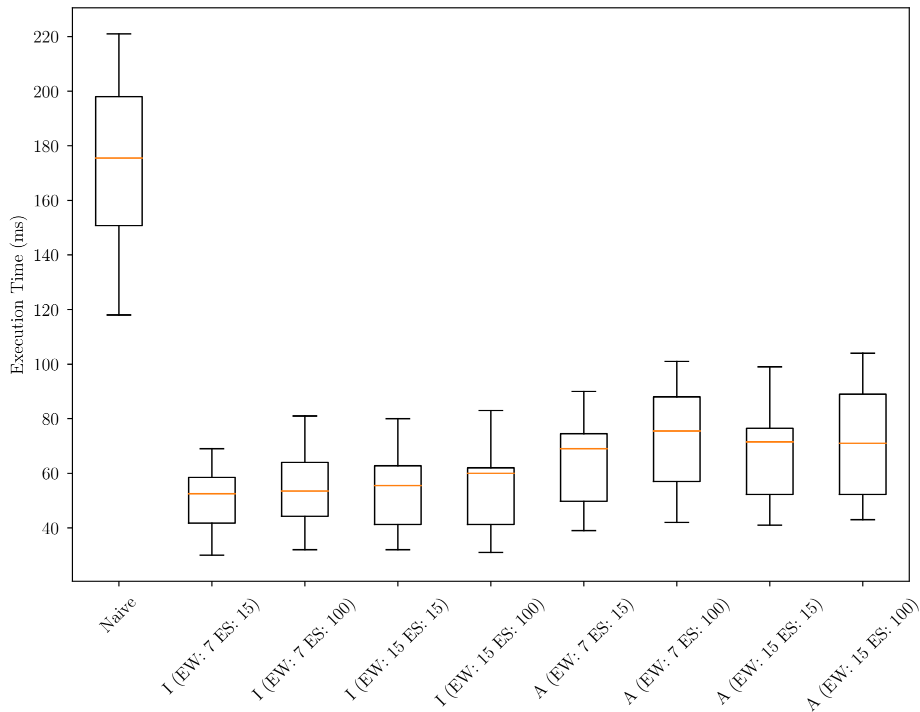


Figure 6.8: Boxplot of the *Spawn* benchmark results with 5.000 fibers

The scalability plot for *Spawn* is presented in Figure 6.9. When compared to Figure 6.8, it is interesting to see that at 5.000 fibers, the results have quite a significant difference. The median of the previous plot for the naive interpreter is about 175 ms, however the mean executing time in the scalability plot is around 50 ms. Another interesting observation is that the interpreters seem to roughly scale equally. This may indicate that the naive interpreter is not as slow as spawning threads as suggested in Figure 6.8. No significant evidence for whether *O3* is satisfied for spawn is present.

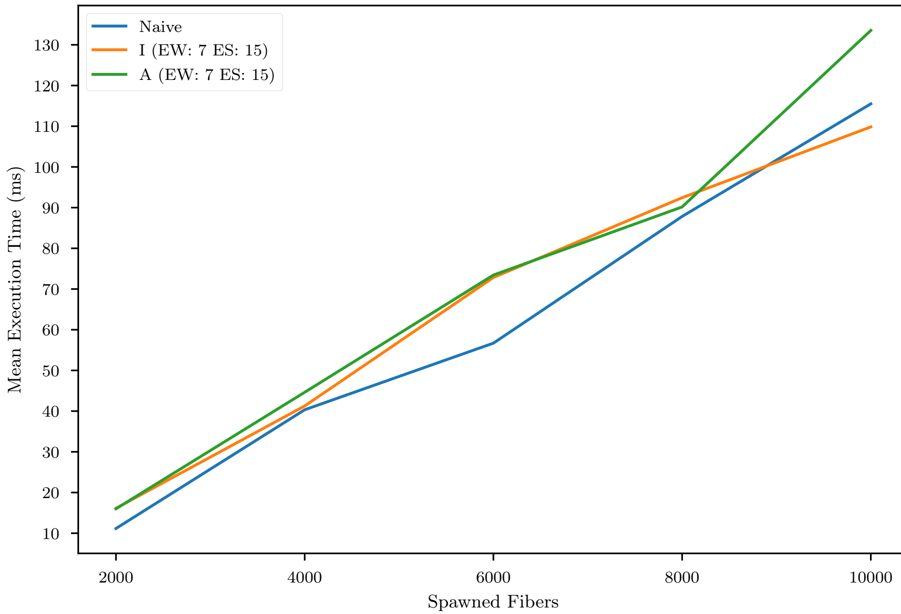


Figure 6.9: Scalability plot of the *Spawn* benchmark showing how the amount of fibers relates to execution time

An overview of the results can be found in Table 6.5.

Benchmark	Best performer	Worst performer	*O3* satisfied
Pingpong	Naive	A (EW: 15 ES: 15)	–
Threadring	A (EW: 15 ES: 100)	I (EW: 15 ES: 15)	Yes
Big	I (EW: 7 ES: 100)	A (EW: 7 ES: 100)	Yes
Bang	I (EW: 7 ES: 100)	Naive	Yes
Spawn	I (EW: 7 ES: 15)	Naive	No evidence

Table 6.5: Overview of the benchmarks results

6.6 Summary

In this chapter, the comparison between the 3 implemented interpreters of naive, intermediate and advanced were compared with the purpose of determining how much the two latter improve upon the naive. First, the evaluation methodology was explained introducing the benchmark suite of *Pingpong*, *Threadring*, *Big*,

Bang and *Spawn* together with measurement metrics and origin. Secondly, the hardware specifications of the Lenovo ThinkPad X1 Carbon 7th generation device that the benchmarks were ran on was introduced. The emphasis was on the CPU with support for 8 threads. It was also mentioned that the device would be running the benchmarks plugged into a power source for stable and optimal performance. Next, the technique used for timing the benchmarks was introduced, with a timer resolution of 100 nanoseconds which is the smallest unit of time that can accurately be measured by the X1 Carbon.

The benchmark suite was then further elaborated upon, by explaining the parameters of each benchmark, that is the amount of fibers that each benchmark would spawn and why, together with how many rounds the benchmark would run for. Each benchmark was then introduced individually, explaining in-depth how it works, what it measures and how accurate measurement of time is achieved.

Finally, the results of the benchmarks were presented, together with the 9 scenarios that each benchmark would be interpreted in 30 times each. It was explained that since the X1 Carbon has support for 8 threads, the intermediate and advanced interpreter would each be ran using 1 and 2 threads per core. In addition, each of these configurations was ran with 15 and 100 evaluation steps.

The results show that the changes made to the intermediate and advanced interpreters had a negative impact on message sending performance as seen with *Pingpong*, however a positive impact on *Bang*. The negative impact on *Pingpong* makes sense, as the changes were only focusing on how efficient blocked effects were handled, thus adding overhead for messaging. For *Bang* the fibers do not have to stay alive during the whole benchmark and it therefore performs better than *Pingpong*. For benchmarks with reoccurring blocking and context-switching like *Threadring*, the advanced interpreter performs better than the naive and intermediate. For many-to-many messaging programs like *Big*, the intermediate interpreter performed the best as the event sending introduced with the advanced interpreter decreased performance. With the *Spawn* benchmark, it was discovered that the intermediate and advanced interpreters spawn fibers faster than the naive which was expected. In the end, 3 out of 4 benchmarks managed to satisfy the *O3* objective including *Threadring*, *Big* and *Bang*. For *Spawn* the interpreters scaled nearly identically so no strong evidence was found.

The next chapter will introduce some implementation ideas for further enhancement of FIO.

Chapter 7

Future Work

In this chapter, ideas for extensions and further evaluation methods will be discussed for FIO. This includes further improvements for the runtime system, including the workers and interpreters. In addition, for the evaluation, performance and scalability comparisons against other libraries. Below potential ideas for future work is presented.

Improved message passing performance
: As shown in Section 6.5, the performance of message passing is decreasing as the interpreters got more advanced. Albeit expected, it would be a proper next step to see whether it is possible to improve message passing performance while keeping the efficient handling of blocked effects.

Support for concurrent blocking workers
: As stated previously in Section 4.2.3, currently only a single blocking worker is supported for the advanced runtime. Improving the implementation such that multiple blocking workers are able to work together at the same time could potentially increase performance in work loads with large amounts of blocking effects.

Improved effect type
: The current effect type `FIO<'R, 'E>` is only parameterized by the effects success type and failure type. As mentioned previously in Section 4.1.2, an ideal extension to the type could be similar to the environment type of ZIO.

This would also give encouragement for development of environments, for example ZIOs console environment for interacting with the console in a pure way.

Proper cancellation of fibers
The current fiber implementation does not support any kind of cancellation of a fiber. Support for this functionality would be useful if an error has happened and the fiber computation is thus not required. Another example would be the `race` function from the API, where the slower fiber could be canceled.

Improved data structure monitoring and dead lock detection
The data structure monitor and dead lock detector could be improved. More data structures could be monitored during runtime that may be of interest to the user, and for the dead lock detector multiple deadlock scenarios could be added for detecting a wide array of deadlocking scenarios.

Improved syntax by implementing F# computation expressions
The syntax of how the user interacts with the library could be improved significantly by implementing support for F# computation expressions. As mentioned in Chapter 3, FIO (DC) have support for those as shown in Listing 16 from line 5 to 9. This includes the `do!`, `let!` and more operators that would encapsulate functionality like sequencing and succeeding. Computation expressions are similar to Scala's for comprehensions as seen previously in Listing 15.

Additional evaluation
Further evaluation of the library could be made. It would be interesting to attempt to compare the scalability and performance of FIO versus ZIO and Cats Effect to see if there is any significant difference between the libraries. Additionally, it could also be interesting to see if there is a significant difference of CPU and memory usage between the interpreter implementations. It would also be interesting to do more extensive evaluation of how well larger amounts of spawned fibers, evaluation workers and steps scale in regards of performance.

Chapter 8

Conclusion

The purpose of this thesis was to design and implement a type-safe and highly concurrent runtime system with the name of FIO using the F# programming language. Related work in the form of programming toolkits for concurrent applications such as Cats Effect and ZIO were investigated for inspiration of design and functionality. The primary goals of the thesis was to determine if F# is a feasible language for developing a highly concurrent runtime system together with the 3 objectives of *O1:* guaranteeing type-safety of programs, *O2:* providing a developer-friendly API and *O3:* achieving better scalability with green threads compared to OS threads.

The design and implementation phases ended up providing an intuitive, type-safe and simple to use API implemented as an internal DSL for taking advantage of concurrent programming. In other words, it is guaranteed that no type errors are present if a FIO program is able to be compiled which satisfies objective *O1*. In addition, 14 self-explanatory API functions were designed to satisfy objective *O2*. This was achieved by carefully selecting the design pattern of ADTs with type casting for the interpreter structure. This pattern was chosen after attempting to use multiple other design patterns that each had their own significant flaws. In addition, a carefully selected set of opcodes were designed to provide crucial low-level functionality of which the API could use to provide more high-level functions. A low-level fiber for use as a green thread was designed as well together with a fiber handle for the user. During the implemen-

tation of the system, multiple issues in regards to race-conditions and deadlocks were encountered and solved.

In the end, the evaluation results show that an efficient and highly scalable runtime system has been developed with 3 out of 4 benchmarks satisfying the *O3* objective with the intermediate and advanced interpreters. Not only is the runtime system capable of spawning an arbitrary amount of fibers, but the intermediate and advanced interpreters outperform the naive in 4 out of 5 benchmarks in terms of raw execution time. For this reason, it is possible to conclude that a successful project has been designed and implemented in terms of a programming library that enables the user to write type-safe and highly concurrent programs. The library is ready to be used in real world general-purpose programs by F# developers. One may, however, consider further development of the library by getting inspiration from the ideas found in Chapter 7.

Appendix A

API Function Implementation

```
1  let fio<'R, 'E> (func : Unit -> 'R) : FIO<'R, 'E> =
2    NonBlocking (fun _ -> Ok (func ()))
```

Listing 48: `fio` API function implementation

```
1  let succeed<'R, 'E> (result : 'R) : FIO<'R, 'E> =
2    Success result
```

Listing 49: `succeed` API function implementation

```
1  let fail<'R, 'E> (error : 'E) : FIO<'R, 'E> =
2    Failure error
```

Listing 50: `fail` API function implementation

```
let stop<'E> : FIO<Unit, 'E> =
  Success ()
```

Listing 51: `stop` API function implementation

```
let send<'R, 'E> (value : 'R) (chan : Channel<'R>)
↪   : FIO<'R, 'E> =
  SendMessage (value, chan)
```

Listing 52: `send` API function implementation

```
let receive<'R, 'E> (chan : Channel<'R>) : FIO<'R, 'E> =
  Blocking chan
```

Listing 53: `receive` API function implementation

```
let attempt<'R, 'E1, 'E> (eff : FIO<'R, 'E1>)
↪   (cont : 'E1 -> FIO<'R, 'E>) : FIO<'R, 'E> =
    SequenceError (eff.Upcast(), fun res -> cont (res :?> 'E1))
```

Listing 54: `attempt` API function implementation

```
let zip<'R1, 'R2, 'E> (eff1 : FIO<'R1, 'E>) (eff2 : FIO<'R2, 'E>)
↪   : FIO<'R1 * 'R2, 'E> =
  eff1 >> fun res1 ->
  eff2 >> fun res2 ->
  Success (res1, res2)
```

Listing 55: `zip` API function implementation

```
let race<'R, 'E> (eff1 : FIO<'R, 'E>) (eff2 : FIO<'R, 'E>) :
↪  FIO<'R, 'E> =
  let rec loop (fiber1 : LowLevelFiber) (fiber2 : LowLevelFiber)
  ↪  =
    if fiber1.Completed() then fiber1
    else if fiber2.Completed() then fiber2
    else loop fiber1 fiber2
  spawn eff1 >> fun fiber1 ->
  spawn eff2 >> fun fiber2 ->
  match (loop (fiber1.ToLowLevel())
  ↪  (fiber2.ToLowLevel())).Await() with
  | Ok res -> Success (res :?> 'R)
  | Error err -> Failure (err :?> 'E)
```

Listing 56: `race` API function implementation

Glossary

.NET .NET is a free and open-source, managed computer software framework for Windows, Linux, and macOS operating systems. It is a cross-platform successor to .NET Framework. The project is primarily developed by Microsoft employees by way of the .NET Foundation 2, 50, 77

AMD Advanced Micro Devices (AMD) is an American semiconductor company that develops computer processors and related technologies for business and consumer markets 1

Application domain An application domain is the area where some software system is meant to solve a problem. Application domains include embedded software, scientific software, business software and more 17, 18

Concurrent programming Concurrent programming is a form of computing where several computations are executed concurrently, meaning during overlapping time periods, instead of sequentially i, 1, 2, 95

Declarative programming Declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow. Declarative languages attempt to minimize or eliminate side effects by describing what the program must accomplish in terms of the problem domain 14

Erlang Erlang is a general-purpose programming language and runtime environment. Erlang has built-in support for concurrency, distribution and fault tolerance. Erlang is used in several large telecommunication systems from Ericsson 76

F# A functional, general purpose programming language. F# (pronounced f-sharp) programming primarily involves defining types and functions that are type-inferred and generalized automatically i, iii, 1, 2, 6, 14, 19, 21, 22, 24, 27, 29, 31, 44, 94–96

Haskell Haskell is a statically-typed, general-purpose, purely functional programming language with type inference and lazy evaluation. Haskell is mostly used in academia 16, 17, 20

Intel Intel Corporation is an American corporation and technology company. It is the world's largest semiconductor chip manufacturer and is the developer of the x86 series of microprocessors, the processors found in most personal computers 1

Java Java is a high-level, class-based, object-oriented programming language. It is a General-Purpose programming language made for to programmers write code once and run it anywhere thanks to the Java Virtual Machine 17–19

MATLAB MATLAB is a high-performance programming and numeric computing platform used to analyze data, develop algorithms, and create scientific models 17

Message passing Message passing is a technique generally used between computer processes or threads. Two processes can pass messages between each other to exchange information 5, 9, 11, 81, 93

Scala Scala is a strong, statically-typed general-purpose programming language which supports both object-oriented programming and functional programming 1, 2, 12, 16–20, 22, 94

Sequential programming Sequential programming is a form of computing where a program is executed from start to finish without other processing executing, as opposed to concurrent or parallel programming 1

Type inference Type inference is the automatic detection of the type of an expression in a programming language. This task is usually handled by the languages' compiler 6

Abbreviations

ADT Algebraic Data Type 27–29, 31, 39, 44, 48, 49, 73, 95

API Application Programming Interface i, 2, 3, 17–19, 22–28, 33–36, 42–48, 50, 52, 63, 73, 94, 95, 97–99

CPU Central Processing Unit 1, 18, 77, 81, 82, 92, 94

DSL Domain-Specific Language 13, 17–19, 22, 24, 95

FP Functional Programming 14

GADT Generalized Algebraic Data Type 2, 27, 28, 44

GPL General-Purpose Language 13, 17

HTML HyperText Markup Language 17

ISA Instruction Set Architecture 18

JVM Java Virtual Machine 18, 19

KISS Keep It Simple, Stupid 24, 25

OS Operating System 3, 6, 19, 21, 33, 39, 41, 44, 56, 68, 71, 82, 86, 88, 95

UML Unified Modeling Language 17

VM Virtual Machine 18

XML Extensible Markup Language 17

Bibliography

[APR+12] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for erlang/otp, (2012).

[Cha19] Daniel Chambers. Fio. https://github.com/daniel-chambers/FSharp.Control.FIO, March 2019. Accessed: 15-04-2022.

[Cho22] Przemek Chojecki. Moore's law is dead. now what? | built in. https://builtin.com/hardware/moores-law, February 2022. Accessed: 15-04-2022.

[Cora] Microsoft Corporation. Stopwatch class (system.diagnostics) | microsoft docs. https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=net-6.0. Accessed: 21-05-2022.

[Corb] Microsoft Corporation. Stopwatch.frequency field (system.diagnostics) | microsoft docs. https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch.frequency?view=net-6.0. Accessed: 21-05-2022.

[Fow06] Martin Fowler. Internaldslstyle. https://martinfowler.com/bliki/InternalDslStyle.html, October 2006. Accessed: 15-04-2022.

[Fow19] Martin Fowler. Domain-specific languages guide. https://martinfowler.com/dsl.html, August 2019. Accessed: 14-04-2022.

[IS14] Shams Imam and Vivek Sarkar. Savina - an actor benchmark suite, (2014).

[Lar22] Daniel Larsen. recursion - is it possible to recurse on a type hierarchy with distinct generic parameters in f#? https://stackoverflow.com/q/70792048, January 2022. Accessed: 28-04-2022.

[Maia] ZIO Maintainers. Running effects | zio. https://zio.dev/version-1.x/overview/overview_running_effects. Accessed: 22-04-2022.

[Maib] ZIO Maintainers. Summary | zio. https://zio.dev/version-1.x/overview/. Accessed: 20-04-2022.

[Mai22] ZIO Maintainers. Zio - a type-safe, composable library for async and concurrent programming in scala. https://github.com/zio/zio, April 2022. Accessed: 23-04-2022.

[Pet] Tomas Petricek. Tomas petricek - new ways of thinking about programming. http://tomasp.net/. Accessed: 28-04-2022.

[Typa] Typelevel. Basics - cats effect. https://typelevel.org/cats-effect/docs/2.x/concurrency/basics. Accessed: 24-04-2022.

[Typb] Typelevel. Cats effect - the pure asynchronous runtime for scala. https://typelevel.org/cats-effect/. Accessed: 20-04-2022.

[Typc] Typelevel. Cats effect - the pure asynchronous runtime for scala. https://typelevel.org/cats-effect/users/. Accessed: 20-04-2022.

[Typd] Typelevel. Getting started - cats effect. https://typelevel.org/cats-effect/docs/getting-started. Accessed: 11-05-2022.