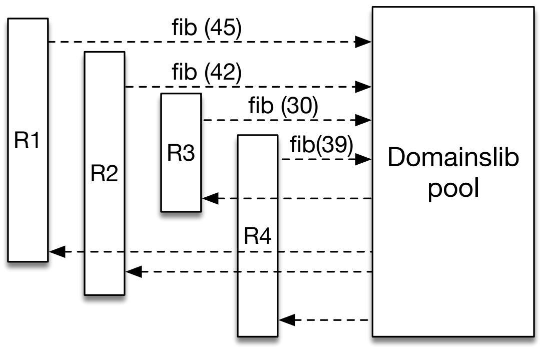
DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI – 600036

Effectively Composing Concurrency Libraries



A Thesis

Submitted by

DEEPALI ANDE

For the award of the degree

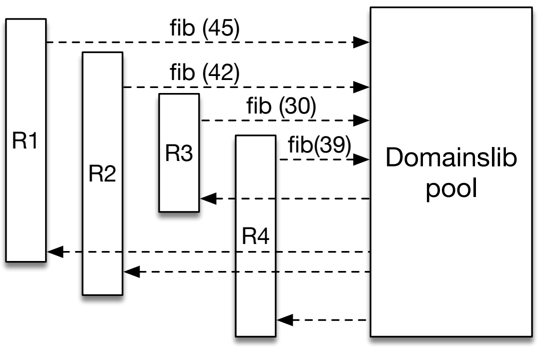
Of

MASTER OF SCIENCE BY RESEARCH

May 2023

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI – 600036

Effectively Composing Concurrency Libraries



A Thesis

Submitted by

DEEPALI ANDE

For the award of the degree

Of

MASTER OF SCIENCE BY RESEARCH

May 2023

To my parents and friends whose constant support and encouragement have helped me to reach for the stars and realize all my aspirations along with a broader perspective on life :-)

THESIS CERTIFICATE

This is to undertake that the Thesis titled **EFFECTIVELY COMPOSING CONCURRENCY LIBRARIES**, submitted by me to the Indian Institute of Technology Madras, for the award of **Master of Science by Research**, is a bona fide record of the research work done by me under the supervision of **Dr. KC Sivaramakrishnan, Dr. Kartik Nagar**. The contents of this Thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Chennai 600036 **Deepali Ande**

Date: May 2023

Dr. KC Sivaramakrishnan
Research advisor
Adjunct Faculty Member
Department of Computer Science & Engineering
IIT Madras

Dr. Kartik Nagar
Research co-advisor
Assistant Professor
Department of Computer Science & Engineering
IIT Madras

LIST OF PUBLICATIONS

I. PRESENTATIONS IN CONFERENCES

Presented a talk on **Composing Schedulers using Effect Handlers** in OCaml Users and Developers Workshop 2022, which was held along with International Conference on Functional Programming (ICFP), 2022.

ACKNOWLEDGEMENTS

I would like to thank my wonderful advisor Dr. KC Sivaramkrishnan sir for his guidance and mentorship throughout this journey of MS research. He taught me the lessons of research very gently and progressively. He gave lot of freedom to think on a solution. We would then discuss all the pros and cons in our weekly meetings. I saw him as a keynote speaker at ICFP and I am still grasping few of things like being confident about your work, talking about it, accepting the feedback and also discussing firmly about our ideas in a very simple terms to a wider knowledgeable audience. These things will not only be useful in my research but anywhere I go. It was because of his guidance, I could present my workshop paper in the OCaml workshop. I also got an experience on how to write a nice research paper. All thanks to KC:-)

I am grateful to Dr. Kartik Nagar sir for agreeing to be my guide and I thank him for all his help in arranging my seminar, thesis meeting. I would like to express my gratitude for my GTC chair Dr. Sukhendu Das sir, GTC member Dr. Nitin Chandrachoodan sir and Dr. Chester Rebeiro sir for their review.

Special thanks to Ms. Sudha Parimala (Senior Software Engineer at Tarides). She has been my goto person to ask doubts about anything related to OCaml. She was always very helpful to me, and we also made a great friendship in this journey. I would also like to thank members of OCaml Labs who gave nice feedback to my research work and also helped me by answering my doubts time to time.

Thanks to all of my friends at IIT Madras, who made my time there memorable, pleasant, and enlightening. I am thankful to my friends for their constant presence, belief in me, and the guidance they have provided throughout my journey not just in academics but in other pursuits as well. . Lastly, grateful to my parents, without their support I could not even come to IIT Madras at the first place.

ABSTRACT

KEYWORDS effect handlers; concurrency; parallelism; scheduler-agnostic; concurrency libraries; suspend; resume.

Effect handlers have proved to be a versatile mechanism for modular programming with user-defined effects. Effect handlers permit non-local control-flow mechanisms such as generators, async/await, coroutines and lightweight threads to be composably expressed. There is increasing interest in supporting effect handlers in industrial-strength languages. The recent major release of OCaml, version 5, supports effect handlers as the primary mechanism for expressing user-level concurrency. Several concurrency libraries have already been designed around effect handlers. However, these libraries are designed monolithically, with their own notion of tasks and mechanisms for inter-task synchronisation. Under this monolithic approach, we face the risk that different concurrency libraries will be incompatible preventing a program from taking advantage of several libraries in the same application. Such is the case with OCaml today with the Lwt and Async libraries with the library ecosystem incompatibly split between the libraries building over either Lwt or Async.

In this thesis, we observe that the composability of effect handlers permits the composability of concurrency libraries. The key idea is to define a uniform yet expressive interface for suspending and resuming tasks, which is implemented by different schedulers. Against this interface, we implement *scheduler-agnostic synchronisation structures* that permit tasks from different concurrency libraries to interact. We also show how to extend this interface to support composition with monadic concurrency libraries such as Lwt and Async. We show how to extend this interface to support various forms of thread cancellation. Finally, we show how this interface helps in safely sharing lazy computations between different concurrency abstractions provided by OCaml including user-level and OS threads.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

OCaml is a powerful functional programming language that has been developed over the last 40 years, offering pragmatic language features that make it ideal for industrial-strength applications About OCaml. Some of its key language features include support for higher-order functions, which are functions that can be passed as an argument or returned as a result, similar to Javascript. Additionally, OCaml supports type inference where we do not need to write explicit types to a variable. It also supports static type checking, which helps to avoid runtime errors and promotes safe code development that can be verified at compile time.

One of the most powerful features of OCaml is its module system, which enables developers to abstract the implementation levels and reuse code in an efficient manner. This module system includes module language, which is concerned with structures that contain collections of definitions, as well as module signatures, which are type declarations. By parameterizing one module structure over another and using functors to return different structures, developers can create highly reusable code that is both efficient and effective.

Another advantage of OCaml is its garbage collector, which frees developers from having to focus on memory management and allows them to concentrate on their application logic. Due to these features, OCaml is used by businesses and organizations, including Meta, Microsoft, Ahrefs, Bloomberg, Jane Street, and Tezos. Additionally, several verification tools like FStar FStar and Coq Coq, and latency-sensitive programs such as mirageOS Mirage OS use OCaml as well.

```ocaml
module type Monad = sig
  (* computation *)
  type 'a t
  (* lift a value to a computation *)
  val return : 'a -> 'a t
  (* sequence two computations *)
  val bind   : 'a t -> ('a -> 'b t) -> 'b t
end
```

Figure 1.1: A Monad Signature

1.1.1 Monadic concurrency libraries

OCaml provides concurrency through the use of monadic, callback-oriented libraries, namely Lwt and Async. However, before delving further into these libraries, it is important to understand what a monad is. Essentially, a monad is any implementation that satisfies the signature in Figure 1.1 along with monad laws Wadler (1992).

Figure 1.1 shows that a value of type `'a` is wrapped around with the monadic return type `'a t` which is modeled as a computation. The `return` function turns any value of type `'a` into a monadic type `'a t`, thus, lifts a value to a computation. The `bind` function binds two computations, where it unwraps the value from first computation and is applied to the provided function. It returns a new monadic type `'b t` as a result.
To illustrate this concept, let's consider an example using Lwt.

Lwt is a library written in OCaml for concurrent IO. Instead of creating multiple system threads to handle blocking IO operations, Lwt uses promises, which can be filled in with value asynchronously. A promise is the primary execution unit in Lwt and is defined by type `'a Lwt.t`. At any given time, a promise can be in the following three states:

- Return x: Promise has been *fulfilled* with the value x.
- Fail exn: Promise has been *rejected* with the exception exn.
- Sleep: Promise is still *pending* and yet to get fulfilled or rejected.

When you have the promise to do some IO, it is not guaranteed to get resolved until the program exits. In order to wait for all the promises in the program to get resolved, we have to include Lwt_main.run function.

```
    Lwt_main.run: 'a Lwt.t -> 'a
```

This function also acts as a scheduler which manages the events received from the outside world to resolve pending promises. (Lwt scheduler comprises of Lwt_engine module, which is a wrapper to the event loops provided by libev, select, etc. and the Lwt_main module Vouillon (2008))

Coming back to the monadic style, Lwt defines a type 'a Lwt.t, which is the type of computation known as a promise.

```
val Lwt.return: 'a -> 'a t
```

Lwt.return v creates a promise that is already fulfilled with value v. It wraps a normal value into a promise.

While the bind function is defined as follows:

```
Lwt.bind : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t
```

Lwt.bind p f creates a promise which waits for pending promise p to be fulfilled by a resulting value. This value is later applied to the function f. Function f when applied with value, returns a promise. Following example will help to understand how bind works.

```
(* Let f is a function which returns int promise *)
val f : unit -> int Lwt.t
let g () =
  let p = f () in
  Lwt.bind p (fun i -> print_int i; Lwt.return ())
```

Here function g creates a promise p by calling function f (). It then waits for promise p to complete and return the integer. This integer is given to print statement and it prints the result. Finally, this function also returns a promise of type unit Lwt.t.

Now that we got the basic working of the Lwt scheduler and promises, we will look into quite a few functions in Lwt that we will use in Chapter 4.

- `Lwt.wait ()`: This function returns the pair of pending promise and its resolver. Promises in the Lwt are actually read-only references. To resolve them, Lwt creates an associated resolver of type `'a Lwt.u`, which can be used to fulfil or reject the corresponding pending promise.

  ```
  val Lwt.wait: unit -> 'a Lwt.t * 'a Lwt.u
  ```

- `Lwt.wakeup r v`: One should use `Lwt.wakeup r v` call to fulfil the pending promise associated with the resolver r with value v.

  ```
  val Lwt.wakeup: 'a Lwt.u -> 'a -> unit
  ```

OCaml program is single-threaded, synchronous, and blocking in nature. Using callbacks, it can implement asynchronous programming to achieve a form of concurrency. In the simplest terms possible, a callback function is a function passed into another function as an argument which is executed later within the outer function. However, writing the code using callbacks leads to a callback hell problem. In callback hell, a function may contain several nested callbacks, making it incredibly challenging to comprehend, edit, or maintain the code. Lwt Vouillon (2008) and Async Async, which supports asynchronous programming in OCaml, provide a monadic, callback-oriented programming model (similar to Javascript). Asynchronous approach is particularly useful when dealing with blocking IO operations, as it allows other tasks to execute while we wait for the IO operation to complete.

In addition to having a callback hell issue, Lwt and Async lack support for backtraces. These monadic libraries split the library ecosystem of OCaml into synchronous and asynchronous code. We cannot use them freely within each other. Thus, there is an incompatibility between Async and Lwt libraries. Ultimately, we end up relying on only a single concurrency library. Thus it is necessary to solve all the problems related to monadic concurrency libraries.

1.1.2 Solution via effect handlers

Effect handlers OCaml Effects, a feature introduced in OCaml 5, offer a cleaner and more powerful solution to the issues faced with monadic concurrency libraries. They offer a mechanism to manipulate non-local control flow in the program using delimited continuations. They are more flexible and easier to use, with the ability to write programs in a direct style. Overall, effect handlers are a significant addition to the OCaml language and have numerous use cases in concurrent programming where tasks are frequently suspended and resumed.

Effect handlers are a generalization of exception handlers, analogous to restartable exception handlers. Computations are described in terms of effects. Say `E` is an effect that is declared using keyword `effect`. The `perform E` operation suspends the current computation, and each performed effect has its corresponding handler to define its meaning, capturing the delimited continuation of the computation. The expressive power of effect handlers comes from the delimited continuation `k`, which captures the suspended computation between the point of perform and the handler. Here, `effect E` has its handler which captures the continuation `k`. The `continue k v` operation resumes the suspended computation and returns the result, `v`, while the `discontinue k Exn` operation resumes an effect by throwing an exception `Exn` at the point of perform, aiding in the cleaning up of resources.

```
1     effect E: string
2
3     let comp () =
4         print_string "0"
5         printf_string (perform E);
6         print_string "3"
7
8     let main () =
9         try comp () with
10        | effect E k -> print_string "1";
11                        continue k "2";
12                        print_string "4"
```

Figure 1.2: Effect handler example

The concept of effect handlers can be illustrated with a simple example 1.2.

Explanation Effect example: First, the keyword `effect` is used to declare an effect named `E`. When effect `E` is performed, it returns a `string` type. Then, the function `comp()` is defined, which is called from the `main()` function. When executed, `comp()` first prints `0`, then performs `effect E`, suspending the current computation. Control is transferred to the handler for `effect E`, which is defined on line 10 in Figure 1.2 and captures the continuation `k`. Inside the handler, the program prints `1` and then resumes the suspended computation by executing `continue k "2"`, which returns `2` to line 5. The program continues with the normal execution, printing `3`, and then finishes executing the handler, printing string `4`.

The final overview can be seen in the following Figure 1.3. Now that we understand how effect handlers work, they play a key role in providing concurrency in OCaml. Concurrency is a programming technique that involves the overlapping execution of tasks, while parallelism focuses on the simultaneous execution of computations to enhance application performance. By leveraging delimited continuations and effect handlers, OCaml offers an excellent choice for implementing concurrent applications. In OCaml, domains serve as the fundamental unit of parallelism, while fibers represent user-level threads designed to achieve concurrency. OCaml 5.0 introduces Eio Eio and

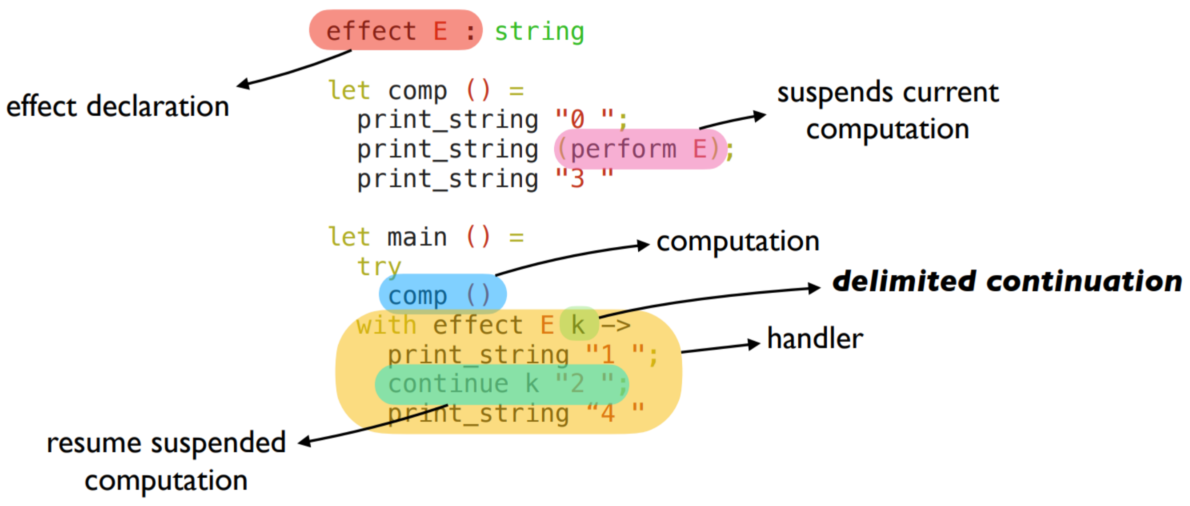

```
effect E : string

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

Figure 1.3: Overview: Effect handler example

Domainslib Domainslib libraries to provide concurrency and parallelism capabilities within applications, respectively. This combination empowers developers to leverage the strengths of both concurrency and parallelism, resulting in efficient and high-performing OCaml applications. Various libraries are built using effect handlers. We will take a look at these libraries in the following section.

1.1.3 Eio

In OCaml, the Unix module Unix uses blocking IO operations, which makes it less suitable for concurrent programs such as network services and interactive applications. To address this limitation, developers have turned to monadic concurrency libraries like Lwt and Async. However, these libraries have their drawbacks, highlighting the need for a standardised, unified IO API in OCaml 5.0. Introducing Eio library, a tailored solution designed for OCaml 5.0, offers an effect-based direct style IO stack. To leverage Eio, one must utilise a backend that runs a suitable event loop for the underlying platform. For Linux, the recommended backend is `io_uring` io_uring which is provided by `eio_linux` package. This backend offers optimised performance compared to previous system calls like `select` or `poll`. By leveraging `io_uring`, Eio enhances the efficiency and responsiveness of IO operations on the Linux platform. For cross-platform compatibility, Eio provides the `eio_posix` package. This utilises the `poll` system call to wait for IO events, making it suitable for various platforms. With Eio and its corresponding backends, OCaml developers gain access to a standardised and efficient IO API, enabling the development of concurrent programs, network services, and interactive applications with improved performance and responsiveness.

What is an event loop? : An event loop is a programming construct that enables the execution of code in a non-blocking and asynchronous manner. It is commonly used in event-driven programming, where the flow of the program is determined by events that occur asynchronously, such as user input or messages from other parts of the system, timers, etc. In a typical event loop, there is a main loop that continuously checks for events and dispatches them to the appropriate event handlers or callbacks. The loop retrieves events from an event queue and processes them one by one. Each event is associated with a specific action or task to be executed when the event occurs. This cycle continues until there are no more events to process. By utilising an event loop, programs can be designed to efficiently handle multiple concurrent events without blocking the execution of other code. Some well-known examples of event loop-based frameworks include Node.js (JavaScript), asyncio (Python), and libuv (C++).

Similalry, in EIO, the `Eio_main.run fn` function is responsible for selecting the appropriate backend for your platform and initiating the respective event loop. Once the event loop is running, it calls the provided `fn` function within it. Within this event loop, various effects encountered in the `fn` function are handled. For example, when an effect like `Fork` is encountered, a new fiber is created, carrying out its own designated task. `Get_context` gets the fiber context. The event loop also handles the Eio library's specific effects, such as `Suspend`, which temporarily suspends the current fiber and saves its fiber context to switch to the next Eio fiber. Additionally, the `Yield` effect allows for seamless transitioning to the next Eio fiber in the sequence. By incorporating these effect handlers within the event loop, Eio facilitates the smooth execution of concurrent operations. The event loop takes care of managing fibers, suspending and resuming their execution, and ensuring efficient task switching, etc.

1.1.4 Domainslib

In OCaml, the domain serves as the fundamental unit of parallelism. The OCaml standard library offers low-level constructs for creating and terminating domains. However, it's important to note that domains are heavyweight entities and correspond directly to OS-level threads, making domain creation an expensive operation. To manage domains effectively while abstracting away the low-level details, the high-level Domainslib library was developed for nested parallel programming. This library facilitates the creation of task pools and the distribution of tasks across available domains. It leverages an efficient work-stealing queue mechanism, allowing idle domains to steal tasks from other domains, ensuring optimal utilisation of resources.

The Domainslib library also provides an async/await mechanism for parallel task execution. Tasks are created using the async function, and await is used to wait for their results. Additionally, Domainslib offers parallel iterators. This library is useful in applications that can parallelise tasks, such as matrix multiplication or recursive computations like the Fibonacci sequence where each recursive computation can be parallelised. By utilising the Domainslib library, developers can harness the power of parallelism in OCaml, enabling efficient execution of tasks and improving performance in scenarios where parallelisation is applicable.

Some of the important functions from library are as follows:

- run p t :

```ocaml
type 'a task = unit -> 'a
val run : pool -> 'a task -> 'a
```

The function `run p t` is designed to execute the task `t` synchronously with the calling domain and the domains in the pool `p`. When the task `t` encounters a blocking promise, the function will execute tasks from the pool `p` until the blocking promise in `t` is resolved. It is recommended to use this function at the top level of your code to encapsulate calls to other functions that await on promises. Failing to do so may result in an `Unhandled` exception being raised by those functions.

- `async p t :`

```
val async : pool -> 'a task -> 'a promise
```

It runs the task `t` asynchronously in the pool `p` and returns the promise `r` which is stores the result of task.

- `await p r :`

```
val await : pool -> 'a promise -> 'a
```

The function `await p r` allows you to wait for the resolution of the promise `r`. While waiting for `r` to be resolved (when the promise is still pending), it calls `perform Wait` effect. The function may execute other tasks from the pool `p` using the calling domain and/or the domains in the pool `p`. This enables concurrent execution of tasks, maximizing efficiency and leveraging the available resources. If the promise `r` is completed successfully, it returns the result. In case of exception, await raises the same exception.

1.1.5 Lazy values

Before delving into the concurrency libraries and their composition, it is also important to have a background for lazy values in OCaml and how it works. Here we will see a simple example of how to create a lazy value with `lazy (expr)`.

```
(* Creating a lazy value using the function lazy*)
let v = lazy (10 + (print_endline "Hello"; 20)) in

(*Now forcing v for the first time*)
let v1 = Lazy.force v in
   print_endline "1: %d" v1;

(* Forcing v for the second time*)
let v2 = Lazy.force v in
   printf "2: %d" v2
```

Figure 1.4: Lazy value evaluation

Lazy functions are a type of deferred computation in which the computation is not evaluated until it is explicitly forced. When the lazy value v is forced for the first time, it will execute the entire expression inside the lazy function and store the final result for all the future `Lazy.force` calls. In the above example in Figure 1.4, when v is forced for the first time, it will print `Hello` and add two numbers. It stores the final result value as 30 and prints it on the console `1: 30`. Forcing the value v a second time simply returns the final value of 30 and prints `2: 30`. Notice that the statement Hello is not printed for the second time.

Having concurrency-safe lazy values is crucial. In the current implementation of lazy values in OCaml, `Lazy.force` raises `Undefined` exception in the following two cases:

- Concurrent forcing: It happens when a lazy value is forced concurrently from different fibers, systhreads or domains. It is important to have concurrency-safe lazy values. Otherwise, there is a potential risk of duplicate computations when two domains force a thunk Thunk simultaneously, where the thunk is not yet evaluated for its result. e.g., when the thunk computes a time-consuming, CPU-intensive task like Fibonacci of 45.

  Another problem is related to memory safety. If two domains mutate the thunk simultaneously, one domain assumes the computation is done and mutates accordingly, while the other domain still forces the thunk. That means both are mutating the same field, which is not correct.

- Self-referencing lazy forcing: When a lazy value tries to force itself recursively, it is referred as self-referencing lazy forcing. For a better understanding, see the below example when the `main ()` function is called.

```
let main () =
    let rec l = lazy (Lazy.force l) in
    Lazy.force l
```

The problem of identifying self-force versus being forced by another domain, thread, or fiber is orthogonal to what we are trying to solve. Assuming we can distinguish between self-force and concurrent force, we address the concurrent forcing problem using a suspend-resume mechanism similar to the one presented in Chapter 5.

1.2 OBJECTIVE AND SCOPE OF THE THESIS

Many modern programming languages provide language-level support for non-local control-flow primitives such as async/await, generators, coroutines, lightweight threads, etc. While some concurrency primitives such as JavaScript generators JSGenerators, C# async/await C# async/await and Kotlin coroutines Kotlin coroutines are implemented with program transformations, there is an increasing trend towards supporting *true* concurrency native in the language. This is because of the inherent cost associated with the program transformation in order to support suspending and resuming tasks as well as the necessity to do something special for supporting features that inspect the program stack such as backtraces and exceptions. These language-level primitives introduce *function colouring* which splits the world between the asynchronous primitives which may suspend execution and synchronous primitives which won't Function Colour.

Alternatively, many languages provide concurrency primitives which are *stackful*, where the runtime system provides support for managing and switching between multiple stacks. They can be broadly split into two groups – one which bakes-in the thread scheduler into the runtime system and the other which permits schedulers to be written as libraries. For example, the Go programming language and GHC Haskell support lightweight language-level threads (called *goroutines* in Go and *threads* in GHC Haskell) which are managed by the runtime system. On the other hand, languages that provide support for schedulers to be written as libraries expose some form of *continuations* at the language level. Java has recently introduced support for virtual threads Java Virtual threads, which is built on top of delimited continuations Loom. The recent release of the OCaml programming language supports effect handlers Sivaramakrishnan *et al.* (2021), which offers a structured primitive for programming with delimited continuations.

There are several downsides to having the scheduler baked into the runtime system as is the case with Go and GHC Haskell. The runtime system not only supports the threads and its scheduler, but also implementations of synchronisation primitives such as locks, condition variables, timers, IO event loop, thread pools, etc. These primitives are often implemented in a low-level language (C in the case of GHC Haskell), which makes it hard to maintain the thread subsystem. This also makes it harder to evolve the language as the changes to the thread subsystem can only be released as part of the language release KC *et al.* (2016).

Moreover, there is no one-size-fits-all scheduling policy for all the programs written in a language. For example, a nested parallel computation such as computing the nth Fibonacci number recursively will benefit from a work-stealing scheduler whereas a scheduler for a web server will benefit from a first-in-first-out (FIFO) scheduling of outstanding requests in order to minimise overall latency. Additionally, each style of application may need their own notion of cancellation of outstanding tasks. For example, a parallel depth first search procedure or a large graph may want to terminate all the outstanding search threads once a result has been found. On the other hand, it is preferable to have *structured concurrency* JEP428 for IO-bound tasks in order to properly clean up resources.

It is with this goal that OCaml 5 introduces effect handlers in order for thread scheduling and concurrency primitives to be implemented as libraries rather than baking it into the runtime. The key idea with effect handlers is to permit effectful operations to be declared and used in a computation without defining how the operations are handled. The meaning of the operations is given by *handlers* of the effects, akin to how exception handlers define how exceptions thrown by a computation are handled. Effect handlers permit handlers of different operations to be expressed modularly and composed together similar to how functions handling different exceptions can be composed together.

The primary motivation to extend OCaml with effect handlers is to permit *direct style* concurrency, as opposed to using monadic concurrency Function Colour libraries such as Lwt Vouillon (2008) and Async Async, where asynchronous functions are represented as monadic computations. Monadic concurrency libraries not only introduce function colours Function Colour that split the API between being synchronous and asynchronous but also often end up specialising the asynchronous APIs to the specific monad, Lwt or Async, due to the lack of higher-kind of polymorphism in OCaml which makes it cumbersome to write code that remains parametric over the concurrency monad. As a result, in OCaml today, one either needs to choose the Lwt or the Async ecosystem and can only use libraries from that ecosystem.

1.2.1 Challenge

The promise of effect handlers is that such an ecosystem split need not happen while also allowing specialisation of schedulers. Indeed, several libraries have already been written utilising effect handlers, which take advantage of the ability to specialise scheduling.

Eio Eio is a library that provides a multicore-capable, direct-style IO stack for OCaml. Eio adopts a work-sharing model where the lightweight user-level tasks, *fibers* in Eio parlance[1],

[1]In this paper, we use *tasks* as a common terminology to represent lightweight threads created and

may be pushed onto other cores, but the tasks remain pinned to the core that they were spawned on. Eio provides structured concurrency JEP428 through a notion of *switches* such that the tasks form a tree-structured hierarchy that ensures that resources are cleaned up deterministically. Eio switches also serve as the notion of task cancellation. When a switch is cancelled, all the tasks that are attached to this switch get cancelled and their resources such as open file descriptors and sockets are closed. Structured concurrency leads to cleaner code and avoids resource leaks. Domainslib Domainslib is a library for nested parallel computation. Unlike Eio, Domainslib uses a work-stealing scheduler that automatically schedules tasks on idle cores. Given the target application, Domainslib neither provides structured concurrency nor does it offer cancellation mechanisms.

While effect handlers permit rich concurrency libraries to be implemented, the fundamental problem is that each of these libraries end up implementing their own incompatible notion of tasks and inter-task synchronisation. Eio provides streams which are bounded queues where taking from an empty stream blocks the Eio task. Domainslib provides async/await mechanism to wait for task completion. Neither of these mechanisms are aware of any other tasks other than the tasks from their own libraries. It is conceivable that an application such as the Tezos blockchain node may want to utilise both of these libraries at the same time, using Eio for the network operations while offloading compute-intensive serialisation and cryptographic primitives to Domainslib. Alas, one cannot build such an application today that utilises both Eio and Domainslib.

This problem of concurrency library composition is not unique to the OCaml ecosystem. Any programming language that provides the ability to implement their own lightweight thread subsystem will need to handle this issue. The Rust programming language provides language-level support for marking asynchronous computations using the `async` keyword. The compiler transforms the async functions into a state machine that represents suspendable computations. Rust does not have a default scheduler for asynchronous tasks

managed by different concurrency libraries and instead relies on libraries called *async runtimes* execute asynchronous applications. As a result, not only does Rust suffer from the problem of function colours, but also suffers incompatibility between asynchronous runtimes Async Rust Book.

1.2.2 Solution

We observe that modularity of effect handlers helps us abstract away from the details of schedulers. For example, in OCaml 5, we can declare the following effects for forking and yielding tasks:

```
type _ Effect.t += Fork : (unit -> unit) -> unit Effect.t
                 | Yield : unit Effect.t
```

The Fork effect takes a thunk which is spawned as a concurrent thread, and the Yield effect yields control to another thread in the scheduler queue. We can define helper functions to perform these effects:

```
let fork (f : unit -> unit) : unit = perform (Fork f)
let yield () : unit = perform Yield
```

The type annotations are not necessary and are only included for clarity. Observe that concurrent programs may call the functions fork and yield without knowing how they are implemented. The implementation is described by each of the concurrency libraries which implement a handler for the Fork and Yield effects.

In this work, we propose a solution for the concurrency library composition problem by describing a single Suspend effect that captures the core details of the threading subsystem. The interface captures the effect of suspending and resuming tasks without appealing to the details of the scheduler implementation. The different concurrency libraries will have to implement a handler only for this effect in order to make them composable with other concurrency libraries.

While our core proposal is astonishingly simple, we show that this one effect is able to capture the complexity of full-fledged concurrent programming support in OCaml. On top of the `Suspend` effect, we show how to implement thread-safe, scheduler-agnostic synchronisation structures such as promises, MVars Peyton Jones *et al.* (1996), mutexes, condition variables, channels, etc. These synchronisation structures may be used to communicate between tasks that belong to different libraries. We show how to extend the `Suspend` effect to capture different, concurrency library specific implementation of task cancellation.

Apart from new libraries such as Eio and Domainslib that take advantage of effect handlers available in OCaml 5, the OCaml ecosystem has millions of lines of legacy code written using monadic concurrency libraries such as Lwt and Async. Hence, it is conceivable that these monadic libraries will continue to be used into the future even when the users switch to OCaml 5. We show how to compose newer effect-based libraries with monadic concurrency libraries using the `Suspend` effect, and thereby enabling an incremental transition of code using monadic concurrency to direct-style. Our solution also helps reconcile the conflict between concurrency and lazy evaluation in OCaml. While OCaml has primitive support for lazy evaluation OCaml Lazy, it is not concurrency-safe.We show how effect handlers enable a graceful solution for concurrency-safe lazy without appealing to a particular concurrency library.

1.3 CONTRIBUTIONS

Our contributions are as follows.

- The design of a single `Suspend` effect that succinctly captures the core details of threading subsystem. Each concurrency library implementing a handler for this effect makes them composable with other concurrency libraries.

- We show how to implement scheduler-agnostic thread-safe synchronisation structures such as IVars, MVars, rendezvous channels, mutex and condition variables on top of this effect without appealing to the details of individual schedulers. These synchronisation structures can be concurrently utilised by tasks from different concurrency libraries.

- We show how to extend the `Suspend` effect in order to enable a variety of task cancellation strategies to co-exist.

- We show how the `Suspend` effect enables the composition of legacy monadic concurrency libraries such as Lwt and Async with effect-based concurrency libraries.

- We illustrate that the `Suspend` effect enables a graceful, backwards-compatible solution for making OCaml lazy values concurrency-safe.

- Our experimental evaluation shows that the composition of concurrency libraries using our solution incurs negligible overheads compared to non-composable alternatives and offers impressive performance improvements in applications where composition is necessary.

The rest of the thesis is organised as follows. The next chapter motivates the need for composition of concurrency libraries, followed by our solution in Chapter 2. We then extend our solution to support task cancellation in Chapter 3. Chapter 4 shows how to compose legacy monadic-concurrency libraries with effect-based ones. Chapter 5 discusses the challenges with lazy evaluation and concurrency and our solution. We evaluate the performance of our solution in Chapter 6. Finally, we discuss the related work in Chapter 7 and offer concluding discussion in Chapter 8.

CHAPTER 2

EFFECTIVE COMPOSITION

2.1 MOTIVATION

In this chapter, we will describe a simplified example that illustrates the need to combine multiple concurrency libraries in the same application. Recall that OCaml 5 brings support for direct-style concurrency using effect handlers and shared-memory parallelism. Suppose the developer wants to implement a highly scalable web server that performs a compute-intensive but parallelisable computation for each request. For edification purposes, let us consider that for each request the server gets a natural number input from the client and the server returns the n^{th} Fibonacci number computed *recursively* back to the client. While the example itself is artificial, it captures a pattern OCaml users have encountered in practice.

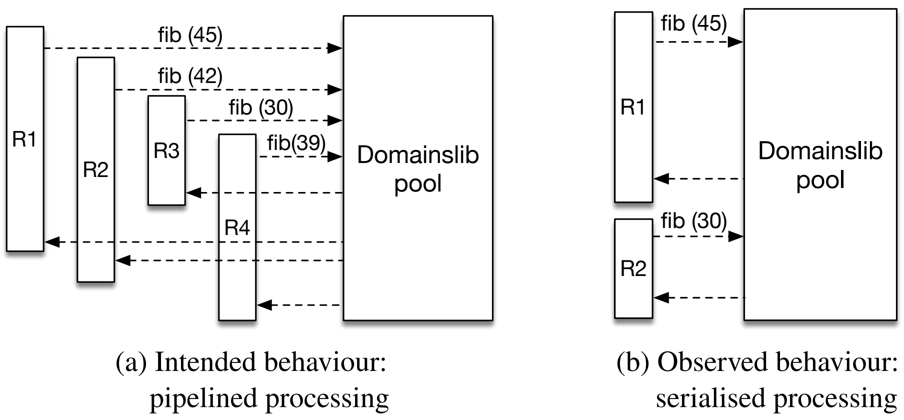
The OCaml ecosystem provides appropriate libraries for implementing different parts of this application. The highly-scalable web server can be implemented with Eio. The nested parallel computation may be parallelised with Domainslib. One may attempt to build the application as shown in Figure 2.1.

Let us walk through the code snippet. The unit of parallelism in OCaml is *domains*. Domains are heavy-weight entities. Each domain maps directly to an OS thread and it is recommended that only as many domains as the number of available cores is created by the user. The library Domainslib permits better management of domains by creating a pool of domains, and submitting tasks to them. In our example, we create a pool of `num_domains` domains. This pool is shared between all of the requests that arrive at the server.

The function `fib_par` computes n^{th} Fibonacci number in parallel using the sequential

```ocaml
module T = Domainslib.Task

(* set up a pool of [num_domains] domains for
   parallel computation *)
let pool = T.setup_pool ~num_domains ()

(* Parallel Fibonacci computation *)
let rec fib_par n =
  let rec fib n =
    if n < 2 then 1
    else fib (n - 1) + fib (n - 2)
  in
  if n > 20 then begin
    let a = T.async pool (fun _ -> fib_par (n-1)) in
    let b = T.async pool (fun _ -> fib_par (n-2)) in
    T.await pool a + T.await pool b
  end else
    fib n

let main () =
  let sock = Eio.Net.listen ... in
  (* Runs once per request in an Eio task *)
  let request_handler n =
    T.run pool (fun _ -> fib_par n)
  in
  while true do
    (* spawn an Eio task to run [request_handler]
       per request *)
    Eio.Net.accept_fork sock ... request_handler ...
  done

let () = Eio_main.run main
```

Figure 2.1: Failed composition of Eio and Domainslib to implement a Fibonacci server.



(a) Intended behaviour: pipelined processing

(b) Observed behaviour: serialised processing

Figure 2.2: Composition of Eio and Domainslib in the Fibonacci server

implementation for small inputs. The `main` function initialises the web server which listens on the socket `sock`. For each connection request, it creates an Eio task that runs the `request_handler` function. All of these Eio tasks are multiplexed on the same domain, but their execution overlaps with the execution of other concurrent requests. The `request_handler` uses the `T.run` function (the same as `Domainslib.Task.run` function) to offload the expensive computation to the Domainslib pool to perform the computation in parallel. The expected behaviour is shown in Figure 2.2a where the concurrent requests are processed in a pipelined fashion.

Unfortunately, the observed behaviour is that the requests are processed in a serialised fashion, one after the other as shown in Figure 2.2b. This is because the function `T.run` is a blocking function that blocks the entire calling domain and not just the Eio task that is making the call. Hence, none of the other tasks from the Eio scheduler can run until `T.run` returns. The fundamental problem is that Domainslib does not have a conception of Eio tasks, and cannot block and unblock them since their semantics is defined by the Eio scheduler. While we can define a point-wise synchronisation solution that works for the composition of Domainslib and Eio, such a pair-wise solution is unsatisfactory as it cannot accommodate other concurrency libraries. What we need is a generic way for tasks from different concurrency libraries to be suspended and resumed without appealing to the specific implementation details of a particular library.

In this chapter, we shall introduce our solution that enables applications such as the Fibonacci server to combine several concurrency libraries developed independently and have them work in the intended fashion. We shall first introduce a simple concurrency-safe scheduler that schedules threads in a FIFO fashion. We shall also use this example as a way to explain the semantics of effect handlers in OCaml 5. We refer interested readers to the OCaml manual page on effect handlers OCaml Effects for a more detailed explanation of their semantics.

2.2 A CONCURRENCY-SAFE FIFO SCHEDULER

As mentioned in Section 1.2.2, the effects `Fork` and `Yield` have been declared but their implementation was not defined. The implementation of these effects are given by the effect handlers in each of the concurrency libraries, which describe how to interpret `Fork` and `Yield`. A computation may `perform` the `Fork` and `Yield` effects without knowing about their implementations.

Figure 2.3 describes the core primitives of the concurrency-safe FIFO scheduler. The scheduler maintains a queue of tasks in a lock-free queue (line 5). A task is a pair of a delimited continuation and the value to resume the continuation with (line 1). We also maintain the number of live tasks in the integer counter `nt` (line 7). We use a mutex and condition variable pair to park the execution of the domain (line 9). It should be noted that mutex and condition from the OCaml standard library operate at the level of domains and not just on the tasks from a concurrency library. Hence, they block and unblock the entire domain and not just the current task from the scheduler.

The `enqueue` function (line 11) takes a delimited continuation and the value to resume the continuation with and pushes a task into the queue. The `dequeue` function is a bit more involved than the `enqueue` function. If the queue is not empty, we resume the next task

```ocaml
type task = Task : ('a,unit) continuation * 'a -> task

let run main =
  (* Lock-free queue *)
  let run_q = Queue.create () in
  (* Number of running tasks *)
  let nt = ref 0 in
  (* Mutex & Condition pair for parking the domain *)
  let m,c = Mutex.create (), Condition.create () in

  let enqueue k v = Queue.push (Task (k,v)) run_q in
  let rec dequeue () =
    match Queue.pop run_q with
    | Some (Task (k,v))-> continue k v
                          (* resume the next task *)
    | None when !nt = 0 -> ()
                          (* No more threads. We're done. *)
    | _ -> (* Some task is blocked elsewhere *)
        Mutex.lock m;
        if Queue.is_empty run_q (* check again with lock *)
        then (Condition.wait c m; Mutex.unlock m; dequeue ())
        else (Mutex.unlock m; dequeue ())
  in

  let rec spawn f =
    incr nt;
    match_with f ()
    { retc = (fun () -> decr nt; dequeue ());
      exnc = begin fun exn ->
        decr nt;
        print_string (Printexc.to_string exn);
        dequeue ()
      end;
      effc = fun (type a) (e : a t) ->
        match e with
        | Yield -> Some (fun (k: (a,_) continuation) ->
            enqueue k (); dequeue ())
        | Fork f -> Some (fun (k: (a,_) continuation) ->
            enqueue k (); spawn f)
        | _ -> None
   }
  in
  spawn main
```

Figure 2.3: Concurrency-safe FIFO scheduler using effect handlers

from the scheduler (line 14). The `continue` primitive resumes a delimited continuation with the given value. Recall that our goal is to build synchronisation structures such as channels and MVars that can be utilised by tasks from different concurrency libraries. As a result, we may encounter the case where the tasks from this scheduler are blocked elsewhere and the queue is empty. In the case when the queue is empty and the `nt` counter is 0, we know that all the tasks created by this scheduler have run to completion. And hence, we are done (line 15). Otherwise, the queue is empty and the `nt` counter is not 0, which indicates that some task created by this scheduler is blocked elsewhere. Hence, we use the mutex and condition variable to park this domain. Care has to be taken to read the queue again with the lock to ensure proper synchronisation.

The `spawn` function (lines 23 to 41) implements the handler for the effects `Fork` and `Yield`. When a task is spawned, we increment the atomic counter `nt`, and evaluate the computation `f` in the context of the effect handler using the `match_with` function (line 25). The computation may return with a unit value (case `retc` on line 26), in which case, we decrement the `nt` counter and run the next tasks from the scheduler. If the task raises an exception (case `exnc` on line 27), then we decrement the `nt` counter, print the exception to standard output and resume with the next task. The case `effc` on line 32 describes how effects are handled. Observe that for each handled effect, we get a delimited continuation `k` that represents the suspended computation from the point of the corresponding `perform`, delimited by the current handler. The handler for `Yield` enqueues the current task and resumes the next one from the scheduler. The handler for `Fork` suspends the current task and recursively calls `spawn` to run the given computation `f` as a new task.

2.3 THE SUSPEND EFFECT

How do we allow the tasks from the scheduler that we have defined to wait on tasks from other schedulers? For this, we need a common way to describe how to suspend and resume tasks. The solution is remarkably simple. We expect concurrency libraries implementing their own schedulers to implement a handler for the following effect.

```
type 'a resumer = 'a -> unit
type _ Effect.t +=
        Suspend:('a resumer -> 'a option) -> 'a Effect.t
```

In order to suspend the current task, the computation performs the effect `Suspend block`, where the function `block` is applied to the `resumer` function that encapsulates the functionality to enqueue the task to the scheduler that it belongs to. In particular, the `resumer` closure will have the delimited continuation of the suspended task in its environment. The synchronisation structures such as channels and MVars can define the function `block` to block the current task. The function `block` is expected to return `None` when the task was successfully blocked. In the presence of multiple domains, it may be the case that the original condition for which the task was about to be blocked already occurred. In which case, the task need not be blocked, and the function `block` should return `Some v`, where `v` of type `'a` is the value necessary to resume the task.

2.4 PROMISES

The use of `Suspend` effect is best understood by looking at how a synchronisation structure might utilise it. To that end, let us implement a concurrency-safe *promise* synchronisation structure that permits tasks from different schedulers to wait for a value. The promise interface is shown below:

```
module type Promise = sig
  type 'a t
  (** The type of promises *)
  val create : unit -> 'a t
  (** Create an unfilled promise *)
  exception Already_filled
  val fill : 'a t -> 'a -> unit
  (** Fill the promise with a value. Raises [Already_filled]
      exception if the promise is already filled. *)
  val await : 'a t -> 'a
  (** If the promise is filled, returns the value in the
      promise. Otherwise, blocks the calling task until
      the promise is filled and returns the filled value. *)
end
```

We can represent the state of the promise as:

```
type 'a state = Full of 'a | Empty of 'a resumer list
type 'a t = 'a state Atomic.t
```

A state of the promise is either full with a value or empty with a list of waiting tasks, represented by a list of resumers. The promise itself is an atomic reference to the state. The promise is created empty:

```
let create () = Atomic.make (Empty [])
```

The `fill` function:

```
exception Already_filled

let rec fill p v =
  let old = Atomic.get p in
  match old with
  | Full _ -> raise Already_filled
  | Empty l ->
      if Atomic.compare_and_set p old (Full v)
      then List.iter (fun r -> r v) l
      else fill p v
```

raises the Already_filled exception if the promise is already filled. Otherwise, it tries to update the atomic reference to the full state. If successful, then the blocked tasks are all resumed using the resumer. It may be the case that the tasks belong to different schedulers. All of the necessary information to resume the task in the right scheduler is encapsulated in the closure.

The await function is the most interesting one.

```
let await p =
  let rec block r =
    let old = Atomic.get p in
    match old with
    | Full v -> Some v
    | Empty l ->
      if Atomic.compare_and_set p old (Empty (r::l))
      then None else block r
  in
  let old = Atomic.get p in
  match old with
  | Full v -> v
  | _ -> perform (Suspend block)
```

If the promise is full, then the await function returns the filled value. Otherwise, the await function performs the Suspend block effect. The scheduler for the current task handles the Suspend block effect and applies block to the resumer r.

The block function reads the atomic location again. If the promise has since been filled, then block returns Some v to the scheduler. The scheduler immediately resumes the same task with the value v. Otherwise, block atomically tries to add the resumer r to the promise state. If successful, the `block` function returns `None` to the scheduler. At this point the scheduler switches to the next task from the scheduler queue or parks the domain if there are no other tasks. Otherwise, the `block` function is retried. See Appendix B.1 for more details about `Atomic.compare_and_set` function.

Observe that the promise implementation does not appeal to the specifics of any particular scheduler, but it allows tasks from different schedulers to interact using the promise, only blocking calling task from the corresponding scheduler. We call such implementations *scheduler agnostic*. We have implemented many common synchronisation structures such as channels, MVar, mutex, condition variables, etc., using a similar strategy. These implementations are compatible with any concurrency library that handles the `Suspend` effect.

2.5 HANDLING SUSPEND EFFECT

Let us now extend our scheduler from Section 2.2 in order to handle the `Suspend` effect.

```
| Suspend block -> Some (fun (k: (a,_) continuation) ->
    let resumer v =
      let wakeup = Queue.is_empty run_q in
      enqueue k v;
      if wakeup then begin
        Mutex.lock m; Condition.signal c; Mutex.unlock m
      end
    in
    match block resumer with
    | None -> dequeue ()
    | Some v -> continue k v)
```

The snippet above is added as one more case in our effect handler in Figure 2.3. The `resumer` function first checks whether the scheduler queue is empty. In this case, domain running the scheduler is parked. We must signal the condition c to wake up the domain. We then enqueue the continuation k to be resumed with the value v to the scheduler queue. Finally, if the domain running the scheduler needs to be woken up, then we signal the condition variable c. Note that if multiple domains race to enqueue into the empty queue, at least one of the domains will see the queue to be empty and will wake up the parked domain.

The argument to the Suspend effect, block, is applied to this resumer. If block returns None, then the continuation k (captured in the resumer) is successfully blocked on the synchronisation structure, and the scheduler resumes the next thread from the scheduler. If Block returns Some v, then we immediately resume the current task with the value v.

The takeaway is that, by handling this one effect Suspend in the concurrency library, the concurrency library becomes compatible with the synchronisation structures such as the promise that we have defined earlier.

2.6 FIXING THE FIBONACCI SERVER

We can use our promise implementation to fix the erroneous behaviour in our Fibonacci server from Section 2.1. Recall that the problem in Figure 2.1 was that the call to T.run in the request_handler function blocks the entire domain and not only the current task. In order to get the pipelined behaviour as shown in Figure 2.2a, we replace the request_handler with the one that follows:

```
let request_handler n =
  let p = Promise.create () in
  ignore (T.async pool (fun _ ->
                              Promise.fill p (fib_par n)));
  Promise.await p
```

Here, we create a promise p per request. The function to compute the n^{th} Fibonacci number is sent to the Domainslib pool using the `T.async` function. Importantly, `T.async` does not block the caller. The promise `p` is filled with the result of the execution of `fib_par n` function. The `request_handler` awaits on the promise `p`, which blocks only the current Eio task. Hence, other Eio tasks are free to run and we get the pipelined behaviour.

There are other real world examples apart from Fibonacci server where Fibonacci computation is used to show the compute intensive workload which can be parallelised using Domainslib. Such examples include a database management system in OCaml 5 that may want to handle concurrent client sessions using Eio while offloading query processing to Domainslib. Thesis also mentions the Tezos blockchain in subsection 1.2.1.

CHAPTER 3

CANCELLATION

Languages that support user-level concurrency make it easy to create millions of tasks with ease. Unlike concurrency through heavy-weight OS threads, lightweight tasks are also cancelled frequently. For example, in a parallel depth-first search in a graph, a large number of search tasks may be created. Once the element that we are looking for is found, all the other search threads will need to be cancelled. Similarly, concurrency libraries such as Eio that allow high-performance I/O prefer structured concurrency JEP428 where the tasks are arranged in a tree-structured hierarchy and cancelling a task ensures that all the tasks as well as their resources are cleaned up. The cancellation mechanisms in different concurrency libraries are bespoke, diverse and are expected to be efficient.

3.1 CANCELLATION CHALLENGES

Given that tasks that are blocked on synchronisation structures may be cancelled, cancellation needs coordination between the concurrency libraries and the synchronisation structures. Without taking cancellation into account, we will observe unintended behaviours. To illustrate this, consider that we extend our scheduler from Section 2 with the ability to cancel tasks with the following API:

```
type handle
val fork : (unit -> unit) -> handle
val cancel : handle -> unit
```

We introduce a type of task handlers `handle`. The `fork` function returns a `handle` rather than `unit`. The function `cancel` marks the task represented by the `handle` to be cancelled. The task may either be currently running, ready to run in the scheduler queue or be blocked on some synchronisation structure. If the task is currently not running, then a cancelled task is guaranteed not to run.

We can implement the functionality by extending the scheduler from Figure 2.3 as follows. We only highlight the important changes here and the full code for the scheduler that supports cancellation is found in the Appendix A.

```
type handle = {mutable cancelled : bool}
let cancel task = task.cancelled <- true
type _ Effect.t += Fork : (unit -> unit) -> handle Effect.t
type task =
    Task: handle * ('a,unit) continuation * 'a -> task
```

The `handle` is represented with a boolean mutable field in a record. `cancel` simply sets this field to `true`. The `Fork` effect now returns the `handle` to the newly created task and the suspended task now carries its handle. We use this `handle` to decide whether to resume a suspended task:

```
let rec dequeue () =
  match Queue.pop run_q with
  | Some (Task (handle, k, v)) -> (* resume the next task *)
      if handle.cancelled then discontinue k Exit
      else continue k v
  ...
```

Instead of unconditionally resuming the task, we now examine whether the `handle` has been cancelled. If so, we resume the continuation by raising the `Exit` exception using the `discontinue` primitive. Discontinuing the continuation on cancellation is essential to ensure that the task stack is unwound freeing any resources such as open file descriptors. If the `handle` is not cancelled, then we resume the continuation `k` with the value `v` as before.

Suppose we use this scheduler with a scheduler-agnostic task-level mutex library `TaskMutex`. Similar to the standard library `Mutex` module, `TaskMutex` allows creation, lock and unlock of the mutex, except that the latter blocks only the calling task and not the calling domain. Without paying attention to cancellation, the combination of a scheduler that supports cancellation with the `TaskMutex` breaks. For example, consider the following code:

```ocaml
module M = TaskMutex;;

run (fun () -> (* main task *)
  let m = M.create () in
  let lu () = M.lock (); M.unlock () in
  M.lock m;
  let t1 = fork lu in (* control switches to t1 *)
  cancel t1;
  (* [t2] waiting behind [t1] to lock the mutex *)
  let t2 = fork lu in
 (* lock gets transferred to [t1] which was cancelled *)
  M.unlock m;
  (* [t2] does not get the mutex, and [run] gets stuck *)
)
```

Here, we create a mutex `m` to synchronise access between concurrent tasks. The function `lu` simply locks and unlocks the mutex `m`. The main task locks the mutex, and spawns a task `t1` which calls `lu`. By looking at the handler for **`Fork`** effect in Figure 2.3, one can see that the `fork` call suspends the current task and switches control to `t1`, which tries to lock the mutex. Given that the mutex is currently held by the main task, `t1` blocks on the mutex and the control switches back to the main task. The main task now cancels `t1` marking its handle as cancelled. Now, the main task creates `t2`, which also runs `lu`. Given that the mutex is still held by the main task, `t2` blocks behind `t1` waiting to lock the mutex.

Finally, the main task unlocks the mutex and runs to completion. The problem occurs here. Without the knowledge that `t1` has been cancelled, the `unlock` call transfers the lock `m` over to `t1` and enqueues `t1` to the scheduler queue. However, `t1` is immediately terminated by `discontinue` when the `dequeue` function examines it at the end of the execution of the main task. Since the mutex is never unlocked by `t1`, `t2` remains blocked forever. Recall that the `run` function will return only when all the tasks run to completion. Hence, the call to `run` function never returns and the execution deadlocks.

One may argue that the problem here is that the call to `lock` should be aware of cancellation and should be protected by an exception handler that handles the `Exit` exception. But observe that the `Exit` exception comes from the scheduler that was independently developed from the `TaskMutex` module. It is not immediately apparent whether documenting that blocking functions may throw exceptions is the appropriate approach. Either way, the libraries should be built to avoid deadlocks even in the presence of buggy code.

3.2 CANCELLATION AWARENESS

We fix this issue by modifying the signature of the `Suspend` effect.

```
type 'a resumer = 'a -> bool (* instead of [unit] *)
type _ Effect.t +=
        Suspend: ('a resumer -> 'a option) -> 'a Effect.t
```

The only change that we introduce is to make the `resumer` return `bool` instead of `unit`. The `resumer` is expected to return `true` if the task was not cancelled and successfully resumed. Otherwise, it returns *false*.

As before, the use of this interface is split between the concurrency library and the synchronisation structure. In the concurrency library, we modify the Suspend handler as follows:

```
| Suspend block -> Some (fun (k: (a,_) continuation) ->
    let resumer v =
      let wakeup = Queue.is_empty run_q in
      enqueue k v;
      if wakeup then begin
        Mutex.lock m; Condition.signal c; Mutex.unlock m
      end;
      not handle.cancelled
    in
    match block resumer with
    | None -> dequeue ()
    | Some v -> if handle.cancelled then continue k v
                else discontinue k Exit )
```

The handler for the Suspend effect here is the cancellation-aware version of the Suspend handler in Section 2.5. There are two changes. First, the `resume` function returns `true` when the task is not cancelled, and `false` otherwise. It might seem strange that we enqueue the continuation `k` to be resumed with the value `v`. But recall that the `dequeue` function first checks whether the `handle` was cancelled, and if so, discontinues the continuation `k` with the `Exit` exception. The second change is that we also check whether the task has been cancelled in the case when `block resumer` returns `Some v`. Before resuming the continuation `k` with `v`, we confirm that the task has not been cancelled. If cancelled, we immediately `discontinue` the continuation.

Figure 3.1 shows the `TaskMutex` implementation that has been made aware of cancellation. The implementation follows ideas similar to the promise implementation from Section 2.4. The only change necessary to make the implementation aware of cancellation is in the

```
module Mutex : Mutex = struct
  type state = Unlocked | Locked of unit resumer list
  type t = state Atomic.t

  let create () = Atomic.make Unlocked

  let lock m =
    let rec block r =
      let old = Atomic.get m in
      match old with
      | Unlocked ->
        if Atomic.compare_and_set m old (Locked [])
        then (Some ()) else block r (* failed CAS; retry *)

      | Locked l ->
        if Atomic.compare_and_set m old (Locked (r::l))
        then None else block r (* failed CAS; retry *)
    in
    perform (Suspend block)

  let rec unlock m =
    let old = Atomic.get m in
    match old with
    | Unlocked -> failwith "impossible"
    | Locked [] -> if Atomic.compare_and_set m old Unlocked
        then () else unlock m (* failed CAS; retry *)
    | Locked (r::rs) ->
        if Atomic.compare_and_set m old (Locked rs)
        then begin
          if r () then
          () (* successfully transferred control *)
          else unlock m (* cancelled; wake up next task *)
        end else unlock m (* failed CAS; retry *)
end
```

Figure 3.1: Task-level mutex implementation that is aware of cancellation.

lines 26 and 27. When we unlock the mutex, we check whether there are pending tasks waiting to lock the mutex. If so, we try to resume them by invoking the resumer `r`. If the resumer `r` returns `true`, then the task associated with this resumer is not cancelled and successfully resumed. Otherwise, if `r` returns `false`, then the task was cancelled and we retry `unlock` to wake up other blocked tasks.

3.3 EAGER AND LAZY CANCELLATION

The cancellation semantics that we have prototyped here may be termed as *lazy* cancellation. When a task is cancelled, we simply mark its handle as cancelled and we wait until it is the task's turn to run in the scheduler in order to terminate it with the `discontinue` primitive. In particular, if the cancelled task was blocked on a synchronisation structure, we require that a matching operation is done on the synchronisation structure that unblocks the task and pushes it into the scheduler queue. An alternative would be *eager* cancellation, where, at the point of cancellation, if the task were blocked on a synchronisation structure, it is removed from the structure eagerly and pushed into the scheduler queue. This is orthogonal to the concerns in this thesis and is a concern of the concurrency library and the synchronisation structure. The `Suspend` effect only expects the `resumer` to return `false` irrespective of whether the task cancellation was eager or lazy. Also check Appendix B.2 to know about the related concept of space leaks.

CHAPTER 4

COMPOSING MONADIC LIBRARIES

So far we have focussed on the composition of concurrency libraries written using effect handlers such as Eio and Domainslib. However, given that effect handlers is a fairly recent addition to OCaml, most of the concurrent code in OCaml today are written in monadic concurrency libraries such as Lwt and Async. These library ecosystems are fairly mature and millions of lines of monadic concurrency code are used everyday. It is likely that monadic concurrency will survive many years into the future.

Given the impossibility of a whole-sale migration of the code written in monadic concurrency to direct-style effect based concurrency, there is the need to ensure that the monadic code can be incrementally migrate to effect-based concurrency. Even if the aim is not to migrate code, legacy applications may want to take advantage of newer libraries. For example, an Lwt application may want to offload compute-intensive computation to a Domainslib pool to take advantage of parallelism. Similarly, an Async application may want to utilise Eio to take advantage of newer OS features for efficient IO such as `io_uring` io_uring. While solutions such as `Lwt_domain` Lwt_domain and Eio bridges to Lwt and Async exist, such point-wise solutions are unsatisfactory. For example, Eio-Lwt bridge cannot take advantage of parallelism since Lwt is not parallelism-safe. Ideally, we would like to run Lwt on one domain and Eio on multiple domains to get the best performance. In this chapter, we show that our solution enables monadic concurrency libraries such as Lwt and Async to be composed with effect-based concurrency libraries.

4.1 MONADIC API FOR SYNCHRONISATION STRUCTURES

There are several challenges to enable such a composition. The notion of a continuation is different between direct-style concurrency libraries based on effect handlers and monadic

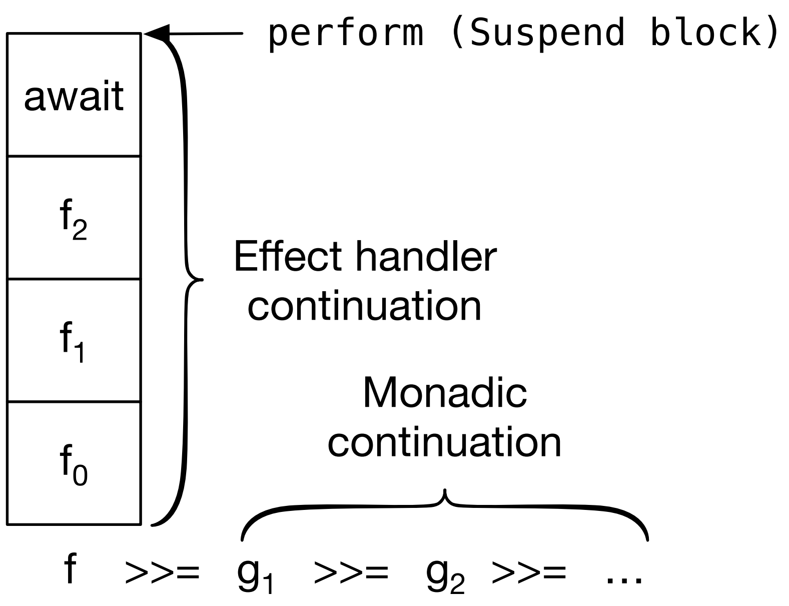


Figure 4.1: Two continuations in a monadic concurrency library.

concurrency libraries. With effect handlers, the continuation is represented by a segment of the call stack Sivaramakrishnan *et al.* (2021) managed by the runtime whereas Lwt and Async essentially utilise callback functions as continuations. This leads to the situation where we will have two orthogonal continuations in the program. For example, consider that the Lwt program uses the promise from Section 2.4. The program state in this case is shown in the Figure 4.1. Since the promise API is in direct-style, the call to the `await` function may include several intermediate function calls `f0`, `f1` and `f2` from the Lwt scheduler. The continuation captured by performing the `Suspend` effect is the *vertical* one that includes the relevant segment of the stack segment of the stack. On the other hand, the continuation in Lwt is the callback function `g1 >>= g2 >>= ....` It is unclear how to capture and reconcile both of these continuations.

Instead, we obviate the need to capture the vertical stack by wrapping the API of the synchronisation structures in a monadic interface as follows:

```
module Lwt_promise : sig
  type 'a t
  val create : unit -> 'a t
  val fill : 'a t -> 'a -> unit
  val await : 'a t -> 'a Lwt.t
end = struct
  type 'a t = 'a Promise.t
  let create = Promise.create
  let fill = Promise.fill
  let await p = Lwt.return (Promise.await p) (* WIP *)
end
```

The only way to use `Lwt_promise.await` is to bind it with the rest of the Lwt computation using >>=. Hence, `Lwt_promise.await` cannot appear in the nested position as in Figure 4.1. Now, we only need to capture the Lwt continuation. Note that `fill` does not block and hence does not need the Lwt wrapper.

4.2 INTEGRATING SUSPEND EFFECT WITH LWT

The `Lwt_promise.await` that we have defined still needs some work. Recall that `await` performs the `Suspend block` effect which needs to be handled. Since we have introduced a monadic wrapper around `await`, the delimited continuation `k` from the effect handler is no longer necessary. But something has to be done to intercept the `Suspend block` effect and apply the `block` function to a suitably prepared `resumer` for Lwt. For this, we exploit the fact that in OCaml, if there are no handlers for an effect `e`, then the exception `Unhandled e` is raised at the point of `perform`. Our final `await` function is as follows.

```
let suspend_monad block =
  let promise, resolver = Lwt.wait () in
  let resumer v = Lwt.wakeup resolver v; true in
  match block resumer with
  | Some v -> Lwt.return v
  | None    -> promise

let await p =
  try Lwt.return (Promise.await p) with
  | Unhandled (Suspend block) -> suspend_monad block
```

The `Lwt_promise.await` function handles the `Unhandled (Suspend block)` effect and applies the `suspend_monad` function to the `block` function. The `suspend_monad` function performs the task similar to the handler for the `Suspend` effect in effect handler based scheduler from Section 2.5. It uses `Lwt.wait` to get a pair of an Lwt's own internal `promise` and a `resolver`. The `resolver` is the other end of the Lwt `promise`. When the resumer is invoked, the promise is filled using `Lwt.wakeup` on the `resolver` with value `v`, which enables the Lwt task to continue. As in the effect handler based scheduler, the `block` function is applied to the `resumer` with the results appropriately handled. For simplicity, the `resumer` function shown here does not handle task cancellation.

Using this solution, we have implemented Lwt-based versions of synchronisation structures that enables Lwt to seamlessly interact with effect handler based concurrency libraries such as Eio.

CHAPTER 5

CONCURRENCY-SAFE LAZY VALUES

So far we have discussed a number of scheduler-agnostic synchronisation structures such as promises and task-level mutexes. These synchronisation structures were implemented as libraries. It turns out that the OCaml language has a primitive feature that can take advantage of the `Suspend` effect to be parametric over the concurrency library. This feature is *lazy values*.

OCaml has built-in support for deferred computations through lazy values. The special syntax `lazy (expr)` returns a lazy value for computing `expr`. Forcing this lazy value computes `expr` and returns its result. The compiler performs certain optimisations that keeps the cost of accessing the result of the already forced lazy value to a minimum. Lazy values are especially useful to model the case where there are many expensive computations but only a few of whose results will be needed, potentially many times.

In OCaml, forcing a lazy value is not concurrency-safe. When a lazy value is concurrently forced, its behaviour is unspecified but OCaml guarantees that there will be no crashes and that the lazy computation is only forced by one of the callers. The recommendation

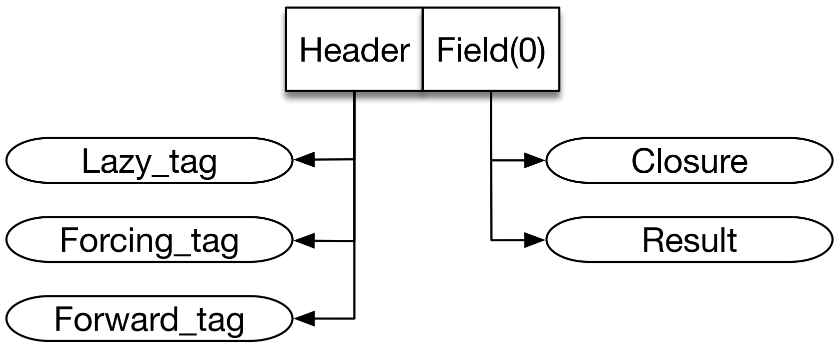


Figure 5.1: Lazy object layout: current

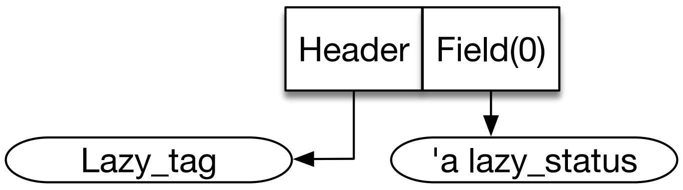


Figure 5.2: Lazy object layout: concurrency-safe

is that any use of lazy should be protected by a mutex. However, the downside of this solution is that even when the lazy computation has been computed, the program will still have to pay for the cost of locking and unlocking the mutex around the lazy value.

What we need is concurrency-safe lazy that can be accessed concurrently without having to resort to the use of a mutex. The idea here is similar to blackholing of thunks in the GHC runtime Harris *et al.* (2005); Marlow *et al.* (2009). Whenever multiple Haskell (lightweight) threads race to evaluate a thunk, the threads that lose the race are blocked on the thunk. When the thunk evaluation completes, the blocked threads are resumed with the result of the thunk evaluation. The difference between GHC and OCaml is that, unlike GHC, OCaml does not have a built-in thread scheduler in the runtime system. Instead, we use the `Suspend` effect to make the lazy implementation parametric over the scheduler.

5.1 LAZY OBJECTS IN OCAML TODAY

In order to enable lazy values to take advantage of `Suspend` effect, we need to modify the layout of lazy values in OCaml. Let us first look at the implementation of lazy values in OCaml today. Figure 5.1 shows the layout of lazy objects in OCaml 5. A lazy value has an object header and one field, each one word in size. Initially, the lazy value has take `Lazy_tag` and the value of the first field is the closure representing the deferred computation. When the lazy value is forced, the tag is first atomically updated to `Forcing_tag`. When the lazy computation either recursively forces itself or another domain concurrency forces the lazy, it will find the tag to be `Forcing_tag`.

In this case, OCaml raises the `Lazy.Undefined` exception. When the computation successfully completes execution, the tag is updated to `Forward_tag` and the first field of the object now points to the result. On the other hand, if the lazy computation raises an exception `exn`, then the first field is updated to a thunk that raises `exn` and the tag is reset from `Forcing_tag` to `Lazy_tag`. This causes subsequent forcing of this lazy value to immediately raise the exception `exn`.

The OCaml garbage collector (GC) performs short-cutting optimisation on lazy value. If the GC observes an object with the `Forward_tag`, it makes the reference to this object directly point to the result. Short-cutting avoids one hop when the program accesses the result of the already evaluated lazy value. When all the reference to the `Forward_tag` object has been short-circuited, the object itself is GCed.

5.2 SUSPENDING ON LAZY

We modify the layout of the lazy object as shown in Figure 5.2 in order to accommodate concurrency. We only use the `Lazy_tag` now and not the other tags. The first field now stores a value of type `'a lazy_state` defined below:

```
type 'a lazy_state =
| Unforced of (unit -> 'a)
| Forcing of int (* unique id *) * 'a resumer list
| Forwarded of 'a
```

The lazy value initially stores `Unforced comp` where the `comp` is the deferred computation. When the lazy is forced, the state is atomically updated to `Forcing (id,[])`. The unique `id` is utilised to distinguish between recursive forcing of the lazy value by `comp` and concurrent forcing of the same lazy by a different task. Recursive forcing is an error and indicates non-termination. This case is similar to GHC Haskell's recursive evaluation of a thunk, where GHC raises the `NonTermination` exception at runtime. In this case, we raise the `Lazy.Undefined` exception. Note that this unique `id` should be unique not only among the tasks from the current scheduler, but also unique between tasks from different schedulers. There are several solutions to this problem, but the problem itself is orthogonal to the focus of this work. Hence, we do not elaborate on this further.

When the lazy is concurrently forced, we use the `Suspend` effect to capture the resumer and atomically add it to the resumer list as in the case of promises. When the computation finishes execution with the result `v`, the state is atomically updated to `Forwarded v` and any tasks that were blocked on this lazy are resumed with the help of the `resumer`. If the computation throws an exception `exn`, the state of the lazy is updated to `Unforced (fun () -> raise exn)`. At this point, we will need to resume the blocked tasks with the `exn`. However, observe that the type of the resumer is a `'a -> bool` function (Section 3.2) and can only be resumed with a value.

5.3 RESUMING WITH AN EXCEPTION

We modify the signature of the `Suspend` effect to be

```
type 'a resumer =
        ('a,exn) Result.t (* instead of ['a] *) -> bool
type _ Effect.t +=
        Suspend: ('a resumer -> 'a option) -> 'a Effect.t
```

where the `Result.t` type defined in the OCaml standard library is:

```
(* module Result *)
type ('a,'e) t = Ok of 'a | Error of 'e
```

This is our final type of `Suspend` effect that we use in our development. The expectation on the `resumer`s is that if the argument is `Ok v` then the task is resumed with the value `v`. If the argument is `Error exn`, then the task is set up such that it continues with the exception exn. Observe that the resumption must be handled in a concurrency-library-specific way. For effect handler based concurrency libraries, we can use the continue and discontinue primitives for these two cases, respectively. In the case of Lwt, the resumer will use Lwt.wakeup and Lwt.wakeup_exn, respectively. Note that Lwt.wakeup_exn exn resolves a promise with the exception exn.

5.4 ADVANTAGES OF THE NEW LAZY DESIGN

Our lazy implementation has a number of nice advantages. The OCaml GC can still short-circuit the lazy values by examining whether the value in the first field was constructed with the Forwarded constructor. Unlike the existing OCaml 5 design, tags in the object header are no longer modified. This eliminates the need for the concurrent GC thread marking the lazy object and the OCaml program modifying the lazy object header Sivaramakrishnan *et al.* (2020).

Thanks to the use of Suspend effect, any concurrency library that handles the Suspend effect can safely share the lazy values. Hence, lazy values can be used by multiple tasks from Eio, Domainslib and Lwt. But what about sharing lazy values between multiple domains (OS threads that run in parallel) and systhreads (OS threads that time-share a domain) OCaml threads library directly when there is no user-level scheduler hosted on them? In this case, we will not have a handler for the Suspend effect. We handle this case similar to the case of handling the Unhandled (Suspend block) in Lwt (Section 4.2) and parking the domain or the systhread on a condition variable. As a result, **our lazy values can be used concurrently by all concurrency abstractions in OCaml** – domains, systhreads, effect handler based concurrency libraries such as Eio and Domainslib and monadic concurrency libraries such as Lwt.

CHAPTER 6

EVALUATION

In this work, we propose to compose concurrency libraries through the single `Suspend` effect. This allows concurrency libraries and synchronisation structures to be implemented independently. In this section, our goal is to show that this approach is pragmatic and does not incur additional overheads compared to implementing bespoke concurrency libraries with their own synchronisation structures.

We have implemented scheduler-agnostic MVars Peyton Jones *et al.* (1996), a general-purpose and expressive synchronisation structure. MVar is a blocking bounded queue with a bound of one. Taking a value from an empty MVar and putting a value into a full MVar blocks the caller. In our experiments, we use these MVars to compose together different libraries. Our experiments are run on the Intel(R) Xeon(R) 5120 CPU x86-64 server with 2 sockets and 28 physical cores. It has 14 cores on each socket and 2 hardware threads per core. Each core runs at a clock speed of 2.20 GHz. The server has 64 GB of main memory. It runs on Ubuntu 20.04. We use OCaml compiler version 5.0.0, which supports effect handlers and shared memory parallelism.

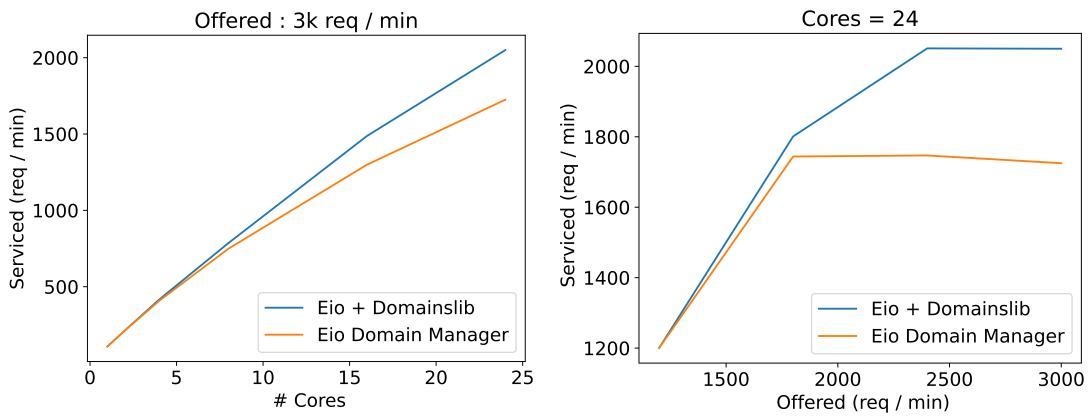
6.1 PRODUCER-CONSUMER BENCHMARK

How does the scheduler-agnostic MVar fare against bespoke synchronisation structures on communication-intensive workloads? In order to answer this, we implemented a

Structure	**Serial**	**Parallel**
Eio stream	1.63	4.39
MVar	1.51	4.15

Table 6.1: Time (in μs) for sending a single message between tasks.

single-producer, single-consumer benchmark in Eio (version 0.6) where the Eio tasks exchange messages. Eio provides streams as an efficient way to synchronise between multiple tasks. Streams are blocking bounded queues that can be used between only Eio tasks. We use the stream bound of 1 in order to match the behaviour of an MVar. Both streams and MVars are concurrency-safe and can be used across domains. They provide similar semantics of suspend and resume, but with the difference that these structures cannot be used outside of Eio's runtime. We run two experiments where the producer and consumer tasks reside on the same domain (serial) and on different domains (parallel). In this experiment, two fibers run a for loop for n times. Out of two fibers, one fiber works as a producer while the other as a consumer. Within each iteration of for loop, the producer increments the variable value. It then sends this value to the consumer via the synchronizing structure (MVar or Eio stream). Similarly, the consumer will read this value from the structure and empties it. Remember, the producer can send the value only when the structure is empty. Otherwise, it suspends its execution, and control goes to the client. Likewise, the consumer will suspend when there is nothing to read, i.e., the structure is empty. In this way, in each iteration, control goes to and fro between the consumer and producer using suspension and resumption. Table 6.1 reports the time in microseconds (μs) to send a single message between the producer and the consumer. When producer and consumer tasks are within the same domain (serial or intradomain), no of iterations n is taken as 10,000,000, and the corresponding latency is shown in column 2. For interdomain communication, latency has increased even for a lesser value of n = 1,000,000. Sending a message between domains is more expensive due to the potential cost of parking and unparking of domains and failed `compare_and_set` operations. In both cases, MVar does not incur any additional overhead compared to other synchronisation structure present in Eio. The results show that our scheduler-agnostic MVar performs on par with Eio streams.



(a) Scaling with cores

(b) Scaling with requests

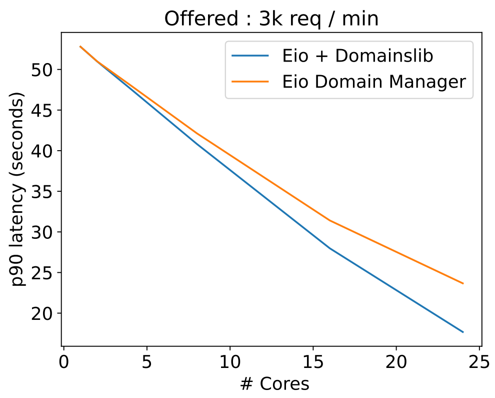
Figure 6.1: Throughput of Fibonacci server.

6.2 FIBONACCI SERVER

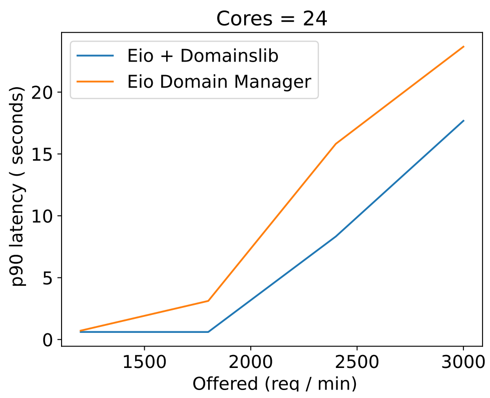
We measure the performance of the Fibonacci server described in Section 2.1. Our server is a full-fledged HTTP server that can handle a large number of concurrent connections and requests. We compare two variants here: (1) **Eio + Domainslib** described in Section 2.6 but uses MVars instead of promises and (2) **Eio Domain Manager** that uses the Eio's built-in support for domains. Note that unlike Eio + Domainslib, the Eio Domain Manager variant does not parallelise a single request to compute the Fibonacci number, but exploits the fact that multiple requests are independent and hence can be run in parallel. Eio Domain Manager variant uses Eio streams for communicating between Eio tasks that may potentially run across different domains. In both cases, the client workload is generated using *wrk2* Wrk2, a high-performance workload generator for testing the performance of HTTP servers. For simplicity, every request computes the 45th Fibonacci number. While our server can accept different inputs, performing the same work in each request allows us to better interpret the experimental results.

6.2.1 Throughput

We perform two experiments and report the results. First we compare the throughput of the different variants. In the first experiment, we maintain a constant load of 3000 requests per minute and vary the number of cores from 1 to 24 in increasing order of



(a) Scaling with cores



(b) Scaling with requests

Figure 6.2: Latency of the Fibonacci server.

powers of 2. We measure the throughput in terms of the requests serviced per minute. The results are presented in Figure 6.1a. In the second experiment, we maintain the core count to be a constant 24 and increase the number of offered requests and measure the serviced request rate. The results are presented in Figure 6.1b. We can see that Eio + Domainslib variant scales better in both experiments due to better parallelisation of available requests.

6.2.2 Latency

We also measure the 90th percentile (p90) latency on the two experiments. As we increase the number of cores from 1 to 24 (like 1, 2, 4, 8, 16, 24) in (Figure 6.2a), the p90 latency comes down as the requests can be parallelised across the available cores. We see that the latency comes down faster with Eio + Domainslib compared to the Eio Domain Manager. This is because with Eio + Domainslib, each of the requests can itself be parallelised and hence the p90 latency for each request comes down faster.

In the second experiment, we fix the number of cores and increase the offered request rate. With increasing number of requests, we expect each of the requests to take longer to complete. Hence, we expect the p90 latency to go up. The results in Figure 6.2b show that the increase is slower in Eio + Domainslib compared to Eio Domain Manager. Thus, in all of our experiments Eio + Domainslib performs better than Eio Domain Manager

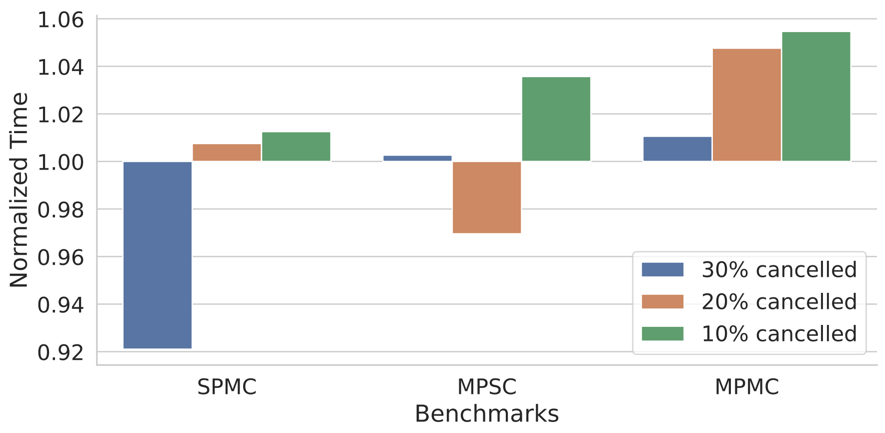


Figure 6.3: Normalized time for Cancellation cost benchmark. Baseline is application without any cancelled tasks

variant.

6.3 CANCELLATION COST

In this section, we measure the cancellation cost using the scheduler that we built in Section 3. We set up the experiment as follows. The benchmark is a producer-consumer benchmark where the producers share 50k items with the consumer. All the producers are run on a domain and all the consumers are run on a different domain. We have three variants of the benchmark: (1) 1 producer and 10k consumers (SPMC), (2) 10k producers and 1 consumer (MPSC) and (3) 5k producers and 5k consumers (MPMC). In each of the variants, we cancel a fixed percentage (10%, 20% and 30%) of tasks. Despite the cancellation, the experiment is set up such that the producers will all together still send 50k items to the consumer. Hence, the total amount of work done by the benchmark is still the same despite cancellation. Given the work done remains the same, the expectation is that, if the cancellation were zero cost, then the total running time variant with and without cancellation will be the same.

The results are presented in Figure 6.3. The graph presents the normalised running time for each of the variants where the baseline is no cancellation. If the cancellation cost were zero, we would expect not to see any bars. The graphs show that the maximum slowdown with cancellation is 5%. But we also see that with cancellations, the programs tend to be faster, and sometimes even faster than the baseline with no cancellations. This is because with cancellation we will have fewer tasks, which can get through the fixed amount of work faster thanks to lower task switching overheads. Overall, the graph shows that cancellation is efficient.

CHAPTER 7

RELATED WORK

Algebraic effect handlers have been an active discipline of theoretical research Plotkin and Power (2001); Plotkin and Pretnar (2009). Many research programming languages and libraries implemented effect handlers. Notable ones are Koka Leijen (2017) and Eff Bauer and Pretnar (2015) which pioneered the scalable programming with effect handlers. OCaml 5 is the first-industrial strength language that offers effect handlers. OCaml uses effect handlers as the primary means of achieving concurrency in direct-style Dolan *et al.* (2018); Sivaramakrishnan *et al.* (2021).

Unlike Koka or Eff, OCaml does not offer *effect safety* – no static guarantee that all the effects performed are handled in the program. Effect safety is an active area of research Hillerstrom *et al.* (2020); Leijen (2017); Biernacki *et al.* (2019*b*,*a*). In OCaml, when there are no handlers for an effect, the `Unhandled` exception is raised at the point of perform. Interestingly, we utilise this behaviour to compose monadic concurrency libraries with direct-style ones.

OCaml 5 introduces language-level support for programming with delimited continuations through effect handlers. Libraries like Eio and Domainslib offer direct-style concurrency using effect handlers as opposed to monadic concurrency libraries such as Lwt and Async. Implementing concurrency libraries over first-class continuations, delimited or otherwise, is a well-studied problem. Several languages in the Lisp family and Standard ML compilers, like SML/NJ and MLton provide support for first-class continuations on top of which concurrency libraries such as Concurrent ML (CML) Reppy (1999) are implemented. CML implements a preemptive FIFO scheduler, provides unbounded blocking channels and mailboxes (MVars) and implements an expressive framework for building communication protocol over events. The novelty in this work is that, unlike the idea of “one concurrency library to rule them all”, the `Suspend` effect permits composition of different concurrency libraries.

Many modern languages provide support for built-in lightweight concurrency such as the Go language and GHC Haskell. Implementing the concurrency support entirely in the runtime system makes the runtime system bloat and monolithic. While Go language is excellent for IO intensive programs, nested parallelism can neither be expressed naturally using goroutines nor is the Go scheduler optimised for nested parallel programming. Sivaramakrishnan et al. KC *et al.* (2016) attempted to relieve GHC Haskell of this problem by implementing support for continuations in GHC and reimplementing the threading subsystem of GHC Haskell completely in Haskell. Similar to OCaml, users may implement their own schedulers for tasks with the help of *scheduler activations*. The threads in GHC Haskell interact non-trivially with other parts of the runtime system including the software-transactional memory, foreign calls, IO manager, lazy evaluation and timers. The runtime system performs *upcalls* to the scheduler in order to service these requests similar to the way the synchronisation structures performing the `Suspend` effect interact with the scheduler, but in the opposite direction. However, compared to our work, Sivaramakrishnan et al. associate a single scheduler with every Haskell execution context (HECs), which are equivalent to domains in OCaml. This prevents richer scheduling policies such as hierarchical scheduling and structured concurrency libraries such as Eio.

Adding support for asynchronous IO for highly scalable concurrent applications to a programming language often creates a split between synchronous and asynchronous code Function Colour. The Rust programming language is no exception. We cannot directly invoke async functions from sync functions in Rust. We cannot therefore arbitrarily combine sync and asynchronous code. The execution of async code requires an async runtime. Rust does not have a built-in async runtime, unlike Go or GHC Haskell.

It has a number of community-maintained crates such as Tokio Tokio, Async-std Async-std, Rayon Rayon, Smol, that offer various async runtimes for concurrency. Similar to OCaml, Rust faces an interoperability problem across different runtimes due to the presence of multiple async runtimes Async Rust Book. Two async runtimes cannot be freely combined. Nonetheless, efforts have been made to establish compatibility layers between Tokio and other runtimes Async Rust Book. It is possible to merge the Tokio and Rayon libraries via a bespoke one-shot channel Rayon Tokio Crate. However, these are not universal solutions for composing runtimes.

Stephen et al. Muller *et al.* (2017) developed a language and graph-based cost model to combine competitive and cooperative threading models. The idea here is to implement a scheduling policy that can handle both competitive and cooperative threads in the same scheduler that guarantees that the theoretical cost bounds developed in the paper are respected by the implementation. Unlike this, our goal is to allow composition of different schedulers developed independently. Each scheduler maintains its own tasks and the tasks from different concurrency libraries interact only through the synchronisation structures.

CHAPTER 8

DISCUSSION

One of OCaml community's strengths is that there are a variety of libraries for each problem. Hence, it is anti-thetical that for concurrent programming, the OCaml programmer had to make a choice between the mutually incompatible Lwt or Async ecosystems. With the arrival of OCaml 5 features, we have another cambrian explosion of concurrency libraries implemented using effect handlers. It is important that the community finds ways for the libraries to coexist so as to prevent futher split in the concurrent programming ecosystem. We believe that our unified interface for expressing concurrency using the Suspend effect:

```
type 'a resumer = ('a,exn) Result.t -> bool
type _ Effect.t +=
        Suspend: ('a resumer -> 'a option) -> 'a Effect.t
```

offers a simple, effective and performant solution for concurrency library composition. We plan to propose the Suspend effect to be included in the OCaml standard library so that different libraries may agree on this effect and develop their composable solutions against this interface. In this work, we also show how to fix the tricky problem of making lazy values compatible with disparate forms of concurrency in the OCaml language using the above interface. We would like to conclude by providing a list of the contributions we accomplished during the course of our research.

APPENDIX A

SCHEDULER WITH CANCELLATION SUPPORT

```
type handle = {mutable cancelled : bool}

let cancel task = task.cancelled <- true

type _ Effect.t += Fork : (unit -> unit) -> handle Effect.t
                 | Yield : unit Effect.t

type 'a resumer = 'a -> bool
(* returns true on successful resumption *)

type _ Effect.t +=
        Suspend: ('a resumer -> 'a option) -> 'a Effect.t

type task =
    Task : handle * ('a,unit) continuation * 'a -> task

let run main =
  (* Lock-free queue *)
  let run_q = Queue.create () in
  (* Number of running tasks *)
  let nt = ref 0 in
  (* Mutex & Condition pair for parking the domain *)
  let m,c = Mutex.create (), Condition.create () in

  let enqueue handle k v =
        Queue.push (Task (handle, k, v)) run_q in
```

```
let rec dequeue () =
  match Queue.pop run_q with
  | Some (Task (handle, k, v)) ->
      (* resume the next task *)
      if handle.cancelled then discontinue k Exit
      else continue k v
  | None when !nt = 0 -> () (* No more threads.
  We're done. *)
  | _ -> (* Some task is blocked elsewhere *)
      Mutex.lock m;
      if Queue.is_empty run_q (* check again with lock *)
      then (Condition.wait c m;
             Mutex.unlock m;
             dequeue ())
      else (Mutex.unlock m; dequeue ())
in

let rec spawn handle f =
  incr nt;
  match_with f ()
  { retc = (fun () -> decr nt; dequeue ());
    exnc = begin fun exn ->
      decr nt;
      print_string (Printexc.to_string exn);
      dequeue ()
    end;
    effc = fun (type a) (e : a t) ->
      match e with
      | Yield -> Some (fun (k: (a,_) continuation) ->
          enqueue handle k (); dequeue ())
```

```
        | Fork f -> Some (fun (k: (a,_) continuation) ->
            let handle' = {cancelled = false} in
            enqueue handle k handle'; spawn handle' f)
        | Suspend f -> Some (fun (k: (a,_) continuation) ->
            let resumer v =
              let wakeup = Queue.is_empty run_q in
              enqueue handle k v;
              if wakeup then begin
                Mutex.lock m;
                Condition.signal c;
                Mutex.unlock m
              end;
              not handle.cancelled
            in
            match block resumer with
            | None -> dequeue ()
            | Some v -> continue k v)
        | _ -> None
  }
in
spawn main
```

APPENDIX B

MISCELLANEOUS

B.1 LOCKFREE PROGRAMMING USING ATOMIC MODULE

When multiple threads or fibers access the same variable at the same time, it causes a data race. To avoid it, we have to synchronise the access to the variable. In such a case, mutex locks can be used. But mutexes in the standard library of the OCaml work at the domain level. They do not work at task level. Thus, if multiple tasks from different domain block on the same mutex, it can cause all the domains to block. In such a case, lockfree programming proves to be a better approach where we do not use any locks. Therefore, we make use of atomic operations. It is important that underlying hardware also supports such an operation. OCaml has an Atomic module to provide these functions. We have used `Atomic.compare_and_set` operations may times in code snippets.

```
val compare_and_set : 'a t -> 'a -> 'a -> bool
```

When we call `Atomic.compare_and_set m seen new`, where m is the mutable atomic reference which is also its current value, it is then set to `new` value only when the `m`'s current value matches with already `seen` value of the reference. If `m` is set to the `new` value i.e. if the operation succeeds it returns *true*, and returns *false* otherwise. In lockfree programming, we retry doing this operation again in case of returning `false` by fetching the latest modified value of the reference. We can see that in the Chapter 2 at multiple places.

B.2 SPACE LEAKS

In Chapter 3 cancellation, we mentioned that our prototype supports lazy cancellation. It also means that the task that is blocked on the synchronisation structure will remain in the structure until corresponding operation unblocks the task. This leads to space leaks Mitchell (2013). Space leak occurs when the program uses more memory than it needs. It is quite different from the memory leak where the leaked memory is never released. But in space leaks the associated memory will be released before the program terminates. At the same time, lazy cancellation gives better efficiency. When the blocked task is cancelled, we do not immediately seek the synchronisation structure to remove the particular task from the structure. Instead we do it lazily. Thus, cost of removing cancelled task has constant time complexity. Otherwise, it takes a linear complexity to search for the cancelled task and remove it. There are ways where we can avoid space leaks. We can implement the *CQS: CancellableQueueSynchronizer* Koval *et al.* (2021) structure for the same. This will remove the cancelled tasks in constant time.

REFERENCES

1. About OCaml (2023). Why ocaml? URL `https://ocaml.org/about`.

2. Async (2023). Typeful concurrent programming. URL `https://opensource.janestreet.com/async/`.

3. Async Rust Book (2023). Asynchronous programming in rust. URL `https://rust-lang.github.io/async-book/01_getting_started/03_state_of_async_rust.html#compatibility-considerations`.

4. Async-std (2023). Async version of the rust standard library. URL `https://docs.rs/async-std/latest/async_std/`.

5. **Bauer, A.** and **M. Pretnar** (2015). Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, **84**(1), 108–123. ISSN 2352-2208. URL `http://www.sciencedirect.com/science/article/pii/S2352220814000194`. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

6. **Biernacki, D.**, **M. Piróg**, **P. Polesiuk**, and **F. Sieczkowski** (2019*a*). Abstracting algebraic effects. *Proc. ACM Program. Lang.*, **3**(POPL). URL `https://doi.org/10.1145/3290319`.

7. **Biernacki, D.**, **M. Piróg**, **P. Polesiuk**, and **F. Sieczkowski** (2019*b*). Binders by day, labels by night: Effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.*, **4**(POPL). URL `https://doi.org/10.1145/3371116`.

8. C# async/await (2023). Asynchronous programming with async and await. URL `https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/`.

9. Coq (2023). The coq proof assistant. URL `https://coq.inria.fr/`.

10. **Dolan, S.**, **S. Eliopoulos**, **D. Hillerström**, **A. Madhavapeddy**, **K. C. Sivaramakrishnan**, and **L. White**, Concurrent system programming with effect handlers. *In* **M. Wang** and **S. Owens** (eds.), *Trends in Functional Programming*. Springer International Publishing, Cham, 2018. ISBN 978-3-319-89719-6.

11. Domainslib (2022). A library for nested parallel programming. URL `https://github.com/ocaml-multicore/domainslib`.

12. Effect example (2023). Talk on retrofitting effect handlers onto ocaml. URL `https://kcsrk.info/slides/retro_effects_hwawei.pdf`.

13. Eio (2022). Effects-based direct-style io for multicore ocaml. URL `https://github.com/ocaml-multicore/eio`.

14. FStar (2023). Proof-oriented programming in f*. URL `https://www.fstar-lang.org/tutorial/book/index.html`.

15. Function Colour (2023). What color is your function? URL `http://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/`.

16. **Harris, T.**, **S. Marlow**, and **S. P. Jones**, Haskell on a shared-memory multiprocessor. *In Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05. Association for Computing Machinery, New York, NY, USA, 2005. ISBN 159593071X. URL `https://doi.org/10.1145/1088348.1088354`.

17. **Hillerstrom, D.**, **L. Sam**, and **R. Atkey** (2020). Effect handlers via generalised continuations. *Journal of Functional Programming*, **30**, e5.

18. io_uring (2023). Efficient io with io_uring. URL `https://kernel.dk/io_uring.pdf`.

19. Java Virtual threads (2023). Jep 444: Virtual threads. URL `https://openjdk.org/jeps/444`.

20. JEP428 (2023). Jep 428: Structured concurrency (incubator). URL `https://openjdk.org/jeps/428`.

21. JSGenerators (2023). function*. URL `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*#description`.

22. **KC, S.**, **T. Harris**, **S. Marlow**, and **S. P. Jones** (2016). Composable scheduler activations for haskell. *Journal of Functional Programming*, **26**, e9.

23. Kotlin coroutines (2023). Coroutines. URL `https://kotlinlang.org/docs/coroutines-overview.html`.

24. **Koval, N.**, **D. Khalanskiy**, and **D. Alistarh** (2021). A formally-verified framework for fair synchronization in kotlin coroutines.

25. **Leijen, D.**, Type directed compilation of row-typed algebraic effects. *In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17. Association for Computing Machinery, New York, NY, USA, 2017. ISBN 9781450346603. URL `https://doi.org/10.1145/3009837.3009872`.

26. Loom (2023). Project loom: Fibers and continuations for the java virtual machine. URL `https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html`.

27. Lwt_domain (2023). Lwt_domain: A library to use domains-based parallelism from lwt. URL `https://github.com/ocsigen/lwt_domain`.

28. **Marlow, S.**, **S. Peyton Jones**, and **S. Singh**, Runtime support for multicore haskell. *In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605583327. URL `https://doi.org/10.1145/1596550.1596563`.

29. Mirage OS (2023). Overview of mirageos. URL `https://mirage.io/docs/overview-of-mirage`.

30. **Mitchell, N.** (2013). Leaking space: Eliminating memory hogs. *ACM Queue*, **11**(9). URL `https://doi.org/10.1145/2538031.2538488`.

31. **Muller, S.**, **U. K. Acar**, and **R. Harper**, Responsive parallel computation: bridging competitive and cooperative threading. *In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017. Association for Computing Machinery, New York, NY, USA, 2017. ISBN 9781450349888. URL `https://doi.org/10.1145/3062341.3062370`.

32. OCaml Effects (2023). Language extensions : Effect handlers. URL `https://kcsrk.info/webman/manual/effects.html`.

33. OCaml Lazy (2023). Manual for lazy module in ocaml. URL `https://v2.ocaml.org/api/Lazy.html#TYPEt`.

34. OCaml threads library (2023). The threads library. URL `https://v2.ocaml.org/manual/libthreads.html`.

35. **Peyton Jones, S.**, **A. Gordon**, and **S. Finne**, Concurrent haskell. *In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96. Association for Computing Machinery, New York, NY, USA, 1996. ISBN 0897917693. URL `https://doi.org/10.1145/237721.237794`.

36. **Plotkin, G.** and **M. Pretnar**, Handlers of algebraic effects. *In* **G. Castagna** (ed.), *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00590-9.

37. **Plotkin, G. D.** and **J. Power**, Adequacy for algebraic effects. *In Foundations of Software Science and Computation Structure*. 2001.

38. Rayon (2022). A data parallelism library for the rust programming language. URL `https://docs.rs/rayon/latest/rayon/`.

39. Rayon Tokio Crate (2021). Rayon tokio crate. URL `https://github.com/andybarron/tokio-rayon`.

40. **Reppy, J. H.**, *Concurrent Programming in ML*. Cambridge University Press, 1999.

41. **Sivaramakrishnan, K.**, **S. Dolan**, **L. White**, **S. Jaffer**, **T. Kelly**, **A. Sahoo**, **S. Parimala**,

A. Dhiman, and **A. Madhavapeddy** (2020). Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.*, **4**(ICFP). URL `https://doi.org/10.1145/3408995`.

42. **Sivaramakrishnan, K.**, **S. Dolan**, **L. White**, **T. Kelly**, **S. Jaffer**, and **A. Madhavapeddy**, Retrofitting Effect Handlers onto OCaml. *In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450383912. URL `https://doi.org/10.1145/3453483.3454039`.

43. Thunk (2023). Thunk. URL `https://en.wikipedia.org/wiki/Thunk`.

44. Tokio (2022). An asynchronous runtime for the rust programming language. URL `https://tokio.rs/`.

45. Unix (2023). Module unix. URL `https://v2.ocaml.org/api/Unix.html`.

46. **Vouillon, J.**, Lwt: A cooperative thread library. *In Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08. Association for Computing Machinery, New York, NY, USA, 2008. ISBN 9781605580623. URL `https://doi.org/10.1145/1411304.1411307`.

47. **Wadler, P.**, The essence of functional programming. *In Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92. Association for Computing Machinery, New York, NY, USA, 1992. ISBN 0897914538. URL `https://doi.org/10.1145/143165.143169`.

48. Wrk2 (2020). A constant throughput, correct latency recording variant of wrk. URL `https://github.com/giltene/wrk2`.

CURRICULUM VITAE

NAME	Deepali Ande
DATE OF BIRTH	28 June 1997

EDUCATION QUALIFICATIONS

2018	**Bachelor of Engineering (BE)**		
		Institution	Maharashtra Institute of Technology, Pune
		Specialization	Computer Engineering
2023	**Master of Science by Research**		
		Institution	Indian Institute of Technology, Madras
		Specialization	Computer Science & Engineering
		Registration Date	7th September, 2020

GENERAL TEST COMMITTEE

Chairperson	Dr. Sukhendu Das Professor Dept of Computer Science & Engineering, IIT Madras
Guide(s)	Dr. KC Sivaramakrishnan Adjunct Faculty Member Dept of Computer Science & Engineering, IIT Madras
	Dr. Kartik Nagar Assistant Professor Dept of Computer Science & Engineering, IIT Madras
Member(s)	Dr. Chester Rebeiro Associate Professor Dept of Computer Science & Engineering, IIT Madras
	Dr. Nitin Chandrachoodan Professor Dept of Electrical Engineering, IIT Madras