Effect handlers in Links

Steven Chang

4th Year Project Report
Electronics and Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

TODO

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Steven Chang*)

Acknowledgements

Table of Contents

Chapter 1

Introduction

Links [16] is a functional programming language that aims to simplify web programming by providing a unified language for all tiers of a web application, which includes the client-side, server-side, and database. To achieve this, Links translates code into the appropriate languages for each tier, such as JavaScript for the browser, Java for the server, and SQL for the database. Additionally, Links supports features that facilitate web development, including manipulation of the Document Object Model (DOM) and event listeners [2]. Furthermore, Links allows for concurrency on both the client and server-side [2], which enables users to create web applications that can handle multiple tasks simultaneously.

In contrast to conventional front-end technologies, Links offers effect handlers [6] as an abstraction for managing effects such as exceptions, input/output operations, and state changes in a modular and composable way. An effect is defined as an operation that can be performed, while a handler is a function that specifies how to handle those operations [6]. By separating the definition of effects from their implementation, it provides a modular and composable way of handling operations in a program, resulting in more flexible and maintainable code. One of the practical applications of effect handlers is its ability to express concurrency in programming languages.

1.1 Motivations

Previous research on the use of effect handlers for concurrent programming has been mostly focused on the system level, despite the growing popularity of effect handlers in recent years. For instance, Multicore OCaml has provided a concurrency library that is implemented with effect handlers [3]. Links, on the other hand, is the first programming language that can build client-side web applications incorporating concurrency through the use of effect handlers.

Meanwhile, the field of front-end development has experienced a rapid growth over the past three decades, evolving from plain HTML, CSS and JavaScript in 1995, to more powerful libraries like jQuery [22] in 2006, and then to modern front-end frameworks such as React [21] in 2013, followed by Vue [23] a year later. Nowadays, these

frameworks are exploring new possibilities, with Vue introducing concurrency [24] and React exploring functional programming paradigms. Specifically, React developers have been inspired by algebraic effects and have made progress integrating similar features into the framework, such as React Hooks [20]. However, these features are more of a mimic of the effect handler structure, as JavaScript does not yet fully support this feature.

Hence, compared with the other functional programming languages and modern front-end frameworks mentioned above, Links has its natural advantage as a functional programming language that is designed for building web applications and already supports the implementation of the effect handler. As a result, this project aims to explore the possibility of effect handlers using Links in the context of web development. The primary focus of this project is to explore concurrent programming with effect handlers, which is becoming an important aspect of modern front-end development with all popular frameworks aiming to integrate concurrent mode into their production workflow.

1.2 Objectives

Previous work has shown that effect handlers in Links can be used to enable concurrency with user-level threads and schedulers [14]. However, there is still room for improvement in terms of the user-friendliness and accessibility of concurrent programming with effect handlers in Links. This project aims to enhance this aspect by allowing users to easily and conveniently define and switch their own handlers within the client code.

To achieve this, the primary objective of this study is to implement various web applications and user-level thread schedulers using Links and effect handlers. The existing code that visualizes user-level threads as lines rendered on the canvas [14] will serve as a starting point for the project. Each web application will feature different line rendering behavior, and users will be able to switch the effect handlers for scheduling threads with minimal changes to the client code.

1.3 Contributions

In this project, I adapted from the existing code base racing_line.links [14] and accomplished the following objectives:

- Improved the user experience of racing_line.links
- Implemented user-level threads that support three different priorities
- Separated the code into UI, client, and scheduler sections for improved separation of concerns
- Extracted the effect interface and user-level thread data structure into separate modules

- Designed and implemented three different types of schedulers, each with a unique enqueue and dequeue mechanism
- Designed and implemented three different applications for visualizing user-level threads
- Updated the scheduler to enable the pause, resume and change of priority of the user-level threads

With these contributions, I have successfully demonstrated a proof of concept demo that allows users to plug in different effect handlers into the client code with minimal effort. This highlights the feasibility of encapsulating effect handlers as independent modules and the ability to plug in different effect handlers for the same computational context without needing to rewrite the entire client code. Compared with the old coding style where users had to replace a large portion of code to change the application behavior, the new structure offers a more efficient and flexible approach by keeping the client code generic when changing effect handlers for different behaviors.

1.4 Report structure

The remainder of the report is structured as follows:

Chapter 2 provides an introduction to modern web development techniques and an overview of the Links programming language, including its syntax and relevant features. It also covers a discussion of effect handlers and concurrent programming using effect handlers, along with related work to provide context for the project.

Chapter 3 outlines the design choices for different scheduling mechanisms that are implemented with effect handlers, as well as applications with different rendering behaviors.

Chapter 4 provides a comprehensive explanation of the modifications made to the existing code and the implementation of each scheduler and application.

Chapter 5 evaluates the final work and discusses the challenges encountered during the development process.

Chapter 6 concludes the work and suggests possible future improvements.

Chapter 2

Background

2.1 Modern web development

The emergence of modern web development technologies marked a significant change in the evolution of the World Wide Web. With the advent of HTML, CSS, and JavaScript, these three technologies formed the backbone of modern web development. Hypertext Markup Language (HTML) [9] provided the basic structure and content of web pages, while Cascading Style Sheets (CSS) [25] is used to add styling to web pages, such as font, colors, and layout. JavaScript [10], on the other hand, enables the creation of interactive and responsive web applications. The integration of these three technologies has paved the way for the creation of functional and engaging websites that could be accessed from any device, anywhere in the world.

One of the very important and fundamental concepts in web development is the Document Object Model (DOM) [4]. It is an essential feature that allows developers to interact with HTML or XML documents dynamically. The DOM is a tree-like structure where each node represents an element in the document. With the DOM, developers can dynamically change the style, content, and structure of a web page in response to user interactions or other events.

Another fundamental concept in web development that is used closely with DOM is event listeners [26] which allow developers to write code that can respond to user events on a web page. An event is simply an action that happens in the browser, such as clicking, scrolling, or hovering over an element. By accessing these elements through the DOM and adding event listeners, developers can detect these events and trigger specific behaviors in response. This functionality allows web applications to respond to user input in real-time, enhancing the overall user experience.

In contrast to HTML, Links uses XML to create web pages and has its own set of DOM operations for XML documents and event listeners using l-event attributes [2]. Further details about the key features of Links will be explained in the next section.

2.2 Links

In traditional web development, developers are required to be proficient in a variety of programming languages, including HTML, JavaScript, Java, Python, and SQL, to build full-stack applications. For beginners, this can be particularly challenging, and the process of linking these languages together can be overwhelming. Moreover, as developers attempt to integrate the frontend, backend, and database, they often encounter the impedance mismatch problem [2], which arises when data must be converted to the corresponding acceptable data types between each tier. This can be a significant limitation for developers seeking to build modern, efficient web applications.

Links was developed to address these issues. As a strict, typed functional programming language, it aims to overcome the challenges of mastering multiple programming languages and the impedance mismatch problem in web development. It provides a single source code for all tiers of a web application, including client-side, server-side, and database. However, due to its research-oriented nature, there are limited resources available for learning Links, and most of the available resources are in the form of example codes and Links Wiki page [17] on GitHub. This section aims to introduce some of the essential features of Links to enhance understanding of this project.

2.2.1 XML and DOM

In contrast to HTML, Links employs XML notations to construct web pages [2]. The syntax for the XML notations is similar to HTML, with the tag name enclosed in `<#>...</#>` syntax. The XML document is maintained by the client as a tree data structure, which is comparable to the DOM. Links offers two types of operations: DOM, which is mutable, and XML, which is for inspection only [2]. It also provides a set of conversion functions to convert custom data types to XML and insert them into the selected nodes. The full operations and functions reference can be accessed through the Links documentation [13].

2.2.2 Event Listeners

In order to enable interaction with the DOM, Links has provided syntax for defining event listeners in the XML code using attributes known as `l-event` attributes [13]. The format of these attributes is `l:name`, with `l:` serving as the prefix and `name` representing the event name. These `l-event` attributes must be assigned with a function that will be executed when the event is triggered [2]. For example,

```
<button l:onclick="onClick()">Click me</button>
```

This indicates the `onClick()` function is invoked when the `button` element is clicked. The complete list of event listeners available in Links can be found in the Links documentation [13].

2.2.3 Foreign Function Interface

In the context of web development, Links has the ability to call JavaScript functions directly within the Links program through the use of Foreign Function Interface (FFI) [15]. The FFI is made possible because modern web browsers run on JavaScript engines, and Links code on the client side is ultimately compiled into JavaScript code during runtime. By providing this feature, it offers extensive flexibility during the development process, enabling smooth collaboration between Links and JavaScript features, opening up new opportunities for developers.

A quick walk through on how to use JavaScript FFI in Links can begin by defining a function in a JavaScript file. The following code illustrates an example of how this can be done:

```
// js/log.js
function _logMessage(msg) {
    return console.log(msg);
}

var logMessage = LINKS.kify(_logMessage);
```

In the above example, `_logMessage(msg)` is a standard JavaScript function that logs the value of the `msg` variable in the console of the browser when it is called. This function is then wrapped inside a conversion function called `LINKS.kify` [15], which produces another function called `logMessage` that can be accessed in the Links program.

On the Links side, the JavaScript functions can be imported as a module (`Log`) for the entire program to access. To achieve this, the `alien` keyword is used in the module block that specifies the external language (`javascript`) and the file path of the JavaScript function. The following code block shows an example:

```
// log.links
module Log {
  alien javascript "js/log.js" {
    logMessage : (String) ~%~> ();
  }
}
```

Inside the `alien` block, the name and the types of the foreign function should be provided. Then the `logMessage` function can be assessed within the Links program by calling `Log.logMessage("Hello World")`.

2.3 Effect Handler

Algebraic effects handler is a feature in functional programming where the concept of algebraic effects was first introduced by Plotkin and Power [18], and later, algebraic effect handlers was introduced by Power and Pretnar [19]. The purpose of the algebraic effects handler is to structure effectful programs in a modular and compositional way. This is accomplished by separating the effect into operations for expressions and

handlers for implementations (discussed in Section 2.3.1.1 and 2.3.1.2) using delimited continuations [5]. This approach enables the programs to pause, resume, and switch between different computation contexts, allowing them to express complex control-flow operations such as I/O, exception, state management, and concurrency. For consistency, the terms effect and effect handler will be used instead of algebraic effect and algebraic effect handler.

The concept of effect handlers was mostly focused on its theoretical aspects in the early stages of the development. However, in recent years, there has been a growing interest in the applicability of effect handlers. As a result, a variety of implementations of effect handlers have emerged, ranging from entire programming languages to libraries [1, 3, 7, 11].

In the effect handler implementation, an effect is conceptualized as a signature of operations, which is written in continuation-passing style (CPS) [12]. Each operation is like a function, but its implementation will be defined by the effect handler. An operation without a concrete implementation is called abstract operation. The following section will delve into the implementation of effect handlers in Links.

2.3.1 Programming with Effect Handlers in Links

In Links, effect handlers are implemented using CPS translations [8], which is enabled by the fact that the Links interpreter is written in CPS [6]. The fundamental idea is to define a handler function that accepts a single argument, which is the computation context of the program. The handler function then defines the implementation of each effectful operation. The operation accepts two parameters: the actual argument(s) passed into the function, and the captured continuation. The upcoming sections will provide a more detailed explanation of the concept of operations and handlers in Links.

2.3.1.1 Operation

Operation is like a function that will produce an output after being performed [6]. The operation itself does not have any semantical meaning towards the program, and the actual implementations for operation is defined entirely by the handler. By convention, when defining an operation, the name should start with a capital letter and should not be left unhandled in the program or otherwise an error will be raised. In order to perform an operation, the syntax is `do Operation(arg)`.

2.3.1.2 Handler

The syntax for implementing handlers in Links is similar as the switch statement in imperative programming or pattern matching in functional programming:

```
handler h(m) {
    case Op1(p, k) -> # Implementation for Op1
    case Op2(p1, p2,..., pn, k) -> # Implementation for Op2
    case Op3(p1, p2, k) -> # Implementation for Op3
    ...
```

```
    case Return(x) -> x
}
```

The parameter `m` is the computation that is being handled. For every operations that is performeds within the computation `m`, it will be mapped to the corresponding `case` for the execution. One thing to notice here is besides the actual parameter(s) being passed into the operations during performing, the operations mapped on the `case` block in handler has one additional parameter `k` at the end. This is the captured continuation as a function that takes one or more arguments. By invoking `k`, the control flow of the program will be transferred from the handler back to the point in computation `m` where this operation was performed [6]. Lastly, the `Return` statement is an essential part of the handler which will be invoked when the computation `m` finishes.

2.3.2 Code example in Links

A straightforward way to explain the implementation of effect handlers in Links is through a series of examples. We begin with a program that prints two strings ”Hello” and ”World” in sequence. Without effect handler, it is straightforward to implement in Links:

```
fun printMessage() {
    print("Hello\n");
    print("World\n");
}
```

2.3.2.1 Printing with effect handler

In order to integrate effect handler into this example, we can convert the `print` method into an operation called `Print` and then define a handler called `forward` that interprets `Print` effect.

```
fun printMessage() {
    do Print("Hello\n");
    do Print("World\n")
}

fun forward(m) {
    handle(m()) {
        case Print(val, k) -> print(val); k(())
        case Return(x) -> ()
    }
}
```

To execute the function, we can call `forward(printMessage)` and it will yields the same result as previous example:

```
Hello
World
() : ()
```

It first replaces `print` method into `Print` operation inside `printMessage` function, and then when executing the program, `printMessage` is passed to the `forward` handler as the computation `m`. Therefore, by invoking `printMessage` inside the handler, `Print` effects will be performed and captured by the `forward` handler.

The actual implementation for each operations is defined in the `case` patterns inside `forward` handler. In this example, `Print` operation after performing will first be mapped to the `Print` case in the handler where it receives two arguments, `val` as the actual parameter being passed into `Print` operation, and `k` which is the continuation function. In the `Print` case, `print` method is invoked before the continuation function `k`, which means the execution will be in order. Therefore in the program `"Hello"` is printed before `"World"`.

2.3.2.2 Continuation

It might seems unnecessary to convert two lines of code into more complicated structures with effect handlers. However, a powerful feature in effect handler is that the developers can manipulate the functions' order of executions in a program by making use of the continuation `k`.

Now on top of the previous example, if we want to print the same strings but show them in the reverse order without changing the order of invocation in `printMessage`, the continuation in effect handler can be handy when dealing with this issue:

```
fun reverse(m) {
    handle(m()) {
        case Print(val, k) -> k(()); print(val)
        case Return(x) -> ()
    }
}
```

We defined a new handler called `reverse`, and then when handling `Print` effect in the `reverse` handler, the continuation `k` is called first, which means `print(val)` will only be invoked after `k` is finished. By wrapping the program continuation into a variable enables the developer to easily taking control over the call stack of the program. Now if we call `reverse(printMessage)`, the result will be:

```
World
Hello
() : ()
```

2.4 Concurrent Programming with effect handler

actor model

Concurrency in Links

Fiber

2.5 Related work

Chapter 3

Design

Evaluate racing_lines.links code and make modification on the buttons

3.1 Schedulers

3.1.1 Scheduler 1

3.1.2 Scheduler 2

3.1.3 Scheduler 3

3.2 Applications

3.2.1 Application 1

3.2.2 Application 2

3.2.3 Application 3

Chapter 4

Implementation

edit Buttons

4.1 Separation of concerns

4.2 Extract effect interface

4.3 Schedulers

4.3.1 Scheduler 1

4.3.2 Scheduler 2

4.3.3 Scheduler 3

4.4 Applications

4.4.1 Application 1

4.4.2 Application 2

4.4.3 Application 3

Chapter 5

Evaluation

5.1 Analysis of results

5.2 User Experience

5.3 Challenges

5.4 Comparison

Chapter 6

Conclusions and future works

Bibliography

[1] Andrej Bauer and Matija Pretnar. “Programming with Algebraic Effects and Handlers”. *CoRR*, abs/1203.1539, 2012.

[2] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. “Links: web programming without tiers”. *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures.*, pages 266–296, 2006.

[3] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K.C. Sivaramakrishnan, and Leo White. *“Concurrent System Programming with Effect Handlers”*, pages 98–117. Apr. 2018.

[4] dom. “DOM”.

[5] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. “On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control”. *Proceedings of the ACM on Programming Languages*, 1, 10 2016.

[6] Daniel Hillerström. “Handlers for Algebraic Effects in Links”. Master’s thesis, Edinburgh, United Kingdom, 2015.

[7] Daniel Hillerström and Sam Lindley. “Liberating Effects with Rows and Handlers”. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, page 15–27, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. “Continuation Passing Style for Effect Handlers”. 84:18:1–18:19, 2017.

[9] HTML. “HTML”.

[10] JavaScript. “JavaScript”.

[11] Ohad Kammar, Sam Lindley, and Nicolas Oury. “Handlers in Action”. *International Conference on Functional Programming*, pages 145–158, 2013.

[12] Sam Lindley. “Algebraic effects and effect handlers for idioms and arrows”. *WGP 2014 - Proceedings of the 2014 ACM SIGPLAN Workshop on Generic Programming*, 08 2014.

[13] Links. “Docs”.

[14] Links. “racing_lines.links”.

[15] Links-lang. “JavaScript FFI”.

[16] Links-lang. “Links”.

[17] Links-lang. “Links wiki”.

[18] Gordon Plotkin and John Power. “Adequacy for Algebraic Effects”. volume 2030, Jan. 2001.

[19] Gordon Plotkin and Matija Pretnar. “Handling Algebraic Effects”. *Logical Methods in Computer Science*, 9(4), 2013.

[20] React. “Introducing Hooks”.

[21] React. “React”.

[22] John Resig. “jQurey”.

[23] Vue. “Vue”.

[24] Vue. “vue-concurrency”.

[25] W3C. “CSS”.

[26] W3Schools. “JavaScript HTML DOM EventListener”.

Appendix A

Complete code for racing lines demo

A.1 racing_lines.links

A.2 fiberInterface.links

A.3 lineDrawer.links

A.4 scheduler.links