Table of Contents

Chapter 1

Background Information

1.1 Links Overview

In the traditional web development process, developers have to master a myriad of languages: HTML and JavaScript for client-side frontend; Java or Python for server-side backend; and finally SQL for the queries in database. Mastering multiple programming languages and learning how to link them together is absolutely an overkill for beginner developers who want to build simple fullstack applications. Even worse, when attempting to link three tiers (i.e. frontend, backend and database) together, *impedance mismatch* problem [3] arises due to the fact that the transmitted data has to be converted to the corresponding acceptable data types between each tiers.

Links is developed to solve these issues. It is a strict, typed functional programming language that is used to write web applications that eliminates impedance mismatch problem with one single source code. On the other hand, since it is a research oriented academic language, the documentations for learning Links are fairly limited with most of the available resources being posted on GitHub in the form of example codes and Links Wiki page [10]. In this section, some of the important features that aids better understanding of this project will be introduced.

1.1.1 Foreign Function Interface

Links supports developers to call JavaScript functions inside Links programs via *Foreign Function Interface* (FFI). This features reveals the possibility to access Reactjs (React written in JavaScript) into Links. Therefore FFI will be used extensively in the early stage of this project to facilitate Reactjs features in Links as an initial prototype, then the core features and features with the use of effect handlers will gradually be re-implemented with native Links code.

A quick walk through on how to use JavaScript FFI in Links is first define a function in the JavaScript file:

```
// js/log.js
function _logMessage(msg) {
```

```
    return console.log(msg);
}

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

In the above example, `_logMessage(msg)` is a normal JavaScript function that will log the value of `msg` variable in the console of the browser when being called. Then this function will be wrapped in a call to `LINKS.kify` [9] that will produce another function `logMessage` that can be accesed in Links program.

On the Links side, the JavaScript functions will be imported as a module to be accessed by the entire program:

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

The keyword `alien` is used to bring the functions inside its block to the scope of this Links program `log.links` by specifying the external language (in this case `javascript`) and its file path. To access the function `logMessage` in Links, simplly call `Log.logMessage("Hello World")`

1.2 Introduction to Algebraic Effect Handler

Algebraic effects handler is a feature in functional programming where the concept of *algebraic effects* was first introduced by Plotkin and Power in 2003 [11] and later in 2013, *algebraic effect handlers* was introduced by Power and Pretnar [12]. The approach of separating the interpretation of effects to a handler from the implementation provides a way to structure effectful programs in a more modular and compositional approach, and it gives the ability to express complex control-flow operations such as I/O, exception, state management and concurrency etc.

There are a few different ways to implement effect handlers. One of them is to use a monadic style of programming, where side-effects are explicitly passed around as values [8]. Another way is to use a continuation-passing style, where the continuation is used to specify what should happen after the side-effect has been handled [6]. However, the general idea is to define a handler function that takes one argument: the computation context, and then effects with the corresponding implementations are defined inside the handler. Each effects takes two parameters: the actual argument(s) passed into that function and the captured continuation. The implementation for the effects can then manipulates the order of invocation between the continuation and the effect.

1.2.1 Programming with Effect Handlers 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");
}
```

1.2.1.1 Printing with effect handler

In order to integrate effect handler into this example, we can convert the `print` method into an effect called `Print` and then define a handler called `forward` that handles `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
() : ()
```

Before going over the code snippets above, there are two core concepts towards the implementation of effect handler in Links that need to be introduced first for better understanding.

1.2.1.2 Operation

Operation is like a function that will produce an output after discharging [5]. 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 discharge an operation, the syntax is `do Effect(arg)`.

1.2.1.3 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 Effect1(p, k) -> # Implementation for Effect1
    case Effect2(p1, p2,..., pn, k) -> # Implementation for Effect2
    case Effect3(p1, p2, k) -> # Implementation for Effect3
    ...
    case Return(x) -> x
}
```

The parameter `m` is the computation that is being handled. For every operations that is discharged 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 discharging, 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 discharged [5]. Lastly, the `Return` statement is an essential part of the handler which will be invoked when the computation `m` finishes.

1.2.1.4 Code walkthrough

Going back to the print example with effect handler, it first converts `print` method into `Print` effect 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 discharged and captured by the `forward` handler.

The actual implementation for each effects is defined in the `case` patterns inside `forward` handler. In this example, discharged `Print` effect 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` effect, 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"`.

1.2.1.5 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
() : ()
```

1.3 React Overview

React is a JavaScript based front-end framework for building interactive user interfaces [14]. The core concept of React is to compose the components where each component is isolated and independent code block that has its own user interface (UI) and state management system.

Compare with the traditional front-end development using vanilla JavaScript with the case of changing the state and UI of the web-page, developers have to constantly query and manipulate the desired elements in the Document Object Model (DOM) [4] which is a tree-like data representation where each nodes represent a HTML element. React provides extensive syntax sugars that simplify this process with better performance and reusability of code by mobilizing each individual components to manage and maintain their own states and re-render the UI automatically when the state changes.

This section will discuss some of the advanced features of React that make use of the concept of Effect Handler.

1.3.1 Hooks

Even though the concept of React component enables the developers to modularize frontend applications in a more intuitive and cleaner manner, it becomes a pain when it comes to managing the states across multiple independent but related components. As the project scales larger, the ideal structure would be the project is being composed by a lot of small but reuseable components with each of them focus on one specific functionality. However in reality, the components often become cumbersome and bloated with deeply nested components and complicated logic [1]. In other words, it is very difficult to separate and reuse the stateful components due to the inconvenience for sharing logic.

One of the solutions is instead of organizing the logic and data flow *across* the compo-

nents, it would be more suitable to process them *inside* the components [1]. Therefore a better approach to design components is to implement them as pure functions (known as Functional Component) instead of traditional classes as in the Object Oriented Programming (known as Class Component). However, Functional Component does not support React features such as state management and lifecycle methods (functions that will be invoked when components are created, updated and destroyed) due to the fact that pure functions do not produce side effects.

React Hooks is designed to solve these issues by injecting React features and side-effects into Functional Components that enables developers to write React application with pure functions. In the context of Algebraic Effects, React Hooks provide abstractions over the actual implementations of hooks where each hooks (such as `useState`, `useEffect`, `useContext`, etc) are actually effects injected into the functions and handled by React during each render cycles [16].

1.3.2 Fiber

In the older version of React (v15), it uses *stack reconciler* to traverse components and construct the virtual DOM [15] (an auxiliary representation of DOM stored in the memory). Then this reconciler detects the difference between the newly established virtual DOM with the current virtual DOM. If any elements differ, it will then inform the *renderer* to take that new virtual DOM and update the changes on the actual DOM tree for rendering [7].

The reason for the reconciler in React v15 named as *stack reconciler* is that the reconciliation algorithm used to traverse the components and make virtual DOM comparison is purely recursive so the term "stack" refers to the call stack. The problem then arised from this recursive algorithm that the traversal cannot be interrupted once it is start running which leads to the issue of dropped frames.

Consider the refresh rate of mainstream browsers as 60Hz, which means the browsers will get refreshed in every $\frac{1000ms}{60Hz} = 16.6ms$. Therefore dropped frames occurs when the reconciliation takes longer than 16.6ms [7]. As a result, the ability for reconciliation to pause and resume as well as taking priority towards certain updates are strongly desired.

In order to support the functionalities mentioned above during reconciliation, the process has to be asynchronous. Thus, *Fiber* is developed as a replacement for traditional call stack: it is designed specifically for React where each single fibers is a virtual stack frame [2]. In React v16, reconciler is reimplemented to adapt Fiber and introduced a new concept called scheduler [13]. This new architecture allows scheduler to first assign priority on each tasks and then pass them to the reconciler known as *Fiber reconciler*. Compare with recursive traversal, Fiber reconciler uses a loop that can be interrupted by toggling the boolean variable flag. Furthermore, Fiber has the ability to control and customize the behaviour of call stack that allows the program to directly manage the stack frames in memory and execute them based on the needs. These characteristics reveals the possibilities to implement behaviours that involves Effect Handler such as concurrency and error boundaries [2].

Bibliography

[1] Dan Abramov. “Making Sense of React Hooks”, 30 Oct 2018.

[2] Andrew Clark. “React Fiber Architecture”, 08 Oct, 2016.

[3] 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.

[4] MDN Web Docs. “Introduction to the DOM”.

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

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

[7] Karthik Kalyanaraman. “A deep dive into React Fiber”, 14 Mar, 2022.

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

[9] Links-lang. “JavaScript FFI”.

[10] Links-lang. “Links wiki”.

[11] Gordon Plotkin and John Power. “Algebraic Operations and Generic Effects”. *Applied Categorical Structures*, 11(1):69–94, 2003.

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

[13] React. “Design Principles”.

[14] React. “Tutorial: Intro to React”.

[15] React. “Virtual DOM and Internals”.

[16] Reese Williams. “Algebraic Effects for React Developers”, 01 Nov, 2018.