Masaryk University
Faculty of Informatics

Concurrency support for PureScript

Master's Thesis

Jan Dupal

Brno, Fall 2016

Replace this page with a copy of the official signed thesis assignment and a copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jan Dupal

Advisor: Jan Obdržálek

Acknowledgement

I would like to express gratitude to my supervisor Jan Obdržálek, for his invaluable advices and patience and John De Goes for suggesting the thesis topic.

Also I would like to thank my family for support and Tomáš and Debbie for proofreading.

Abstract

PureScript is a small strongly typed programming language that compiles to JavaScript. It borrows a number of interesting features, such as type inference, higher kinded polymorphism, extensible effects or type classes, from modern functional languages (e.g. Haskell).

The goal of this thesis is to introduce to PureScript support for concurrency, based on JavaScript technology called Web Workers. The concurrent features should be integrated in a way that is fully compatible with the functional nature of the PureScript language.

Keywords

PureScript, Web Workers, Concurrency, Web Applications, JavaScript

Contents

1 Introduction

During the last decade, *web applications* – applications running in web browser – became ubiquitous part of our lives. Be it a web search engine, a social network, an e-mail client, an accounting software, or even a word processor or a spreadsheet – all of these services can be accessed via web browser running on desktop computers, laptops, or mobile devices.

Thanks to technologies behind web applications we, both as users and developers, do not need to worry whether our *operating system* is supported by a given applications and if we have all the required dependencies installed. All we need is a modern web browser.

However, upswing of web applications comes at a cost – as vendors of web applications attempt to enhance *user experience*, more and more business logic is being moved from application servers to users' web browsers. In order to support such requirements, web browsers provide *client-side scripting* [1]. Client-side scripting usually refers to execution of JavaScript programs in context of web browser with a goal to offer improved *interactivity* to the end user.

With the volume of JavaScript client-side logic growing, developers of web applications face similar problems as in the realm of server-side development. As JavaScript is not a *statically typed* programing language, developer tools for assisting in domain modeling, maintenance, refactoring, and other tasks are as powerful as for statically typed programming languages.

PureScript [2] is a relatively new programming language compiled to JavaScript, which attempts to add *static typing*, but also embraces *functional programming* paradigm in client-side scripting. Being inspired by Haskell programming language it focuses on separation of side-effects and mutable state, while adopting other popular functional programming concepts as *monads*, *type classes* or *extensibe effects*.

While Haskell runtime supports multiple threads for concurrent computation, JavaScript execution models is single-threaded. Such limitation has primarily influence on *responsiveness* of the application, making user experience sluggish.

To alleviate this issue, **Web Workers** [3, 4] JavaScript standard addresses demands to execute long-running background client-side

scripts without negatively affecting the responsiveness of *user interface* – a concept similar to background computation provided by *multitasking*.

The thesis aim is to extend abilities of PureScript by adopting JavaScript technology *Web Workers* in a systematic manner, while honoring philosophy behind PureScript and its ecosystem. The *contribution* is the implementation of `purescript-web-workers` library, which allows PureScript developers conveniently employ *Web Workers* in PureScript applications.

Chapter 2, *Background*, introduces the reader to JavaScript and PureScript programming languages, followed by chapter 3, *Related work*, covering related approaches to multithreading. Chapter 4, *Demonstration assignment*, presents a Web Worker use-case, which advocates the necessity of concurrency in web applications and provides intuition to reason about concurrency in JavaScript. And chapter 5, *Web Workers*, familiarizes reader with *Web Workers* JavaScript standard.

Subsequent chapter 6, *Implementing Web Workers in PureScript*, opens the second part of the thesis by describing goals and requirements for PureScript integration of Web Workers. Following four chapters 7, 8, 9, and 10 describe several approaches to implementing Web Workers to PureScript, each evaluating requirements stated in 6.

The conclusion in chapter 11 summarizes achieved approaches and suggests future work.

2 Background

2.1 JavaScript

In this section we will cover foundation of JavaScript language and its runtime model, which are both essential for the rest of the thesis.

2.1.1 JavaScript as a programming language

JavaScript is a high-level, dynamic, untyped interpreted programming language that is well-suited to object-oriented and functional programming styles. JavaScript derives its syntax from Java, its first-class functions from Scheme, and its prototype-based inheritance from Self. [5]

2.1.2 JavaScript runtime model

The JavaScript runtime model consist of three parts: *heap*, *stack*, and *message queue*. [6] While the *heap* and the *stack* should be familiar from other programming languages, the *message queue* may not.

The *message queue* lists messages to be processed, e.g. user-initiated events (clicks) from the user interface, scheduled executions, asynchronous HTTP responses etc. Each message has an associates function, that is responsible for processing the message. This function is executed when the message is popped from the message queue – this process is called *event loop*. [6]

The *event loop* basically waits for a messages to arrive and then the loop processes the message. Once a message is being processed, the processing block the whole runtime – no other message can be processed during this interval. [6]

Although the event loop may remind us a multitasking scheduler, there is one particular difference – the event loop do not support any kind of *preemption*; a message must be fully processed before the next message in queue takes a turn.

2.2 PureScript

PureScript is a statically-type functional language that compiles to JavaScript. Its primary goal is to provide sane programming language and tooling to cover use cases in web development, that are currently served primarily by JavaScript. [2]

While Haskell, the most notable influencer of PureScript, can be compiled to JavaScript [7] too, PureScript aims to offer a better experience to developer by generating human-readable JavaScript and convenient interoperability with native JavaScript libraries.

2.2.1 Type system

As opposed to JavaScript, PureScript has strict static typing, which is enforced by its powerful type system inspired by Haskell. Inspiration by Haskell means that PureScript provides features like *type classes*, *algebraic data types*, *quantified types* [2] etc.

Although one particular feature of the type system is worth mention – so called *extensible effects*.

While Haskell generally uses `IO` to mark side-effects, PureScript uses a more fine-grained approach, e.g.:

```
Eff (console :: CONSOLE, err :: EXCEPTION) Unit
```

While the `Eff` part roughly corresponds to Haskell's `IO`, the middle part specifying `CONSOLE` and `EXCEPTION` is different – such type represents a side effect, that is allowed to access *browser console* and *throw exception*. However the type signature disallows for example accessing *random number generator*, which has it's own effect type `RANDOM`.

Thanks to better granularity, the developer is able express side-effects in the type signature more precisely.

2.2.2 Foreign Function Interface

Foreign Function Interface, or *FFI*, allows bidirectional interoperability between PureScript and JavaScript. The general approach to FFI is to specify type signature in PureScript and provide JavaScript implementation in a separate file.

Although there are almost no compile-time guarantees, FFI can be considered a reasonable alternative to native PureScript implementation.

For a detailed guide and best practices please refer to the book *PureScript by Example*[2], which is available online free of charge.

3 Related work

3.1 Concurrent Haskell

As PureScript is philosophically rooted in Haskell programming language, it is worth to explore similar problems and their solution in Haskell – *Concurrent haskell*[8] describes how to achieve concurrency in Haskell programming language. The key ideas are: [8]

Processes and their initiation via `forkIO`.

Atomically-mutable state provided by `MVar` variables.

3.2 PureScript Web Worker attempts

To our best knowledge there were two different projects aimed to bring Web Worker support to PureScript.

3.2.1 Fresheyeball/purescript-workers

The project `purescript-workers` was initiated in 2014 and provides `Paralel` module, which provides a set of functions to spawn a worker and communicate. Notably the project uses `Blob` object to spawn a worker script, which is the same approach we've used.

However, the PureScript language has significantly developed since the project was published and the project does not address the *build system integration* requirement covered later in section 6.2.

The project is available on: `https://github.com/Fresheyeball/purescript-workers`

3.2.2 FrigoEU/purescript-webworkers

The project `purescript-webworkers` initiated in 2016 exposes wraper of native JavaScript Web Worker API. Notable feature is `Channel` abstraction built on top of message handlers.

The *build system integration* requirement is not covered.

The project is available on:
`https://github.com/FrigoEU/purescript-webworkers`

4 Demonstration assignment

Before we deep-dive into integrating Web Workers, let us setup an assignment for a demo application. Such assignment should closely reflect needs of real-life usage, while remaining as minimal as possible and allow for incremental changes in the implementation to demonstrate different use-cases of the proposed Web Worker PuresScript API.

In the following chapters, we will iteratively implement this assignment in PureScript in different abstractions, each suitable for a slightly different use-case.

4.1 Assignment

In order to demonstrate the contribution of *Web Workers*, we have designed an assignment to implement PBKDF2 into a simple client-side web application. The application consists of a single button, which triggers CPU-heavy computation of PBKDF2, simulating a long-running task.

PBKDF2 is a password-based key derivation function, as described in [9]:

> A key derivation function produces a derived key from a base key and other parameters. In a password-based key derivation function[1], the base key is a *password* and the other parameters are a *salt value* and an *iteration count*.

Apart from the button, the user interface includes a separate counter, driven by PureScript code periodically increasing the count. This element serves as a visual indicator of the main thread is being blocked by other computation – if any long-running computation is being executed by the main thread, the code responsible for updating the counter will not be scheduled until the long-running task is finished. Such delay will result in freezing the counter.

The described assignment contains essential aspects of a real-life web application:

---

1. For our purposes, PBKDF2 function can be understood as a hashing function.

- User interface logic – represented by the button and the counter
- Business logic with long-running computation – represented by PBKDF2 algorithm

See figure 4.1 – the user interface.

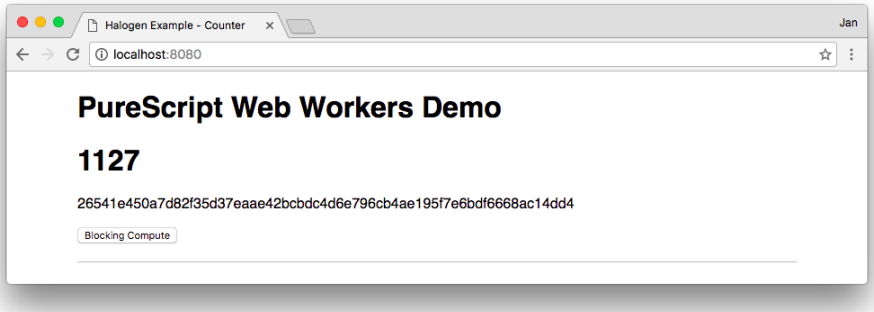

Figure 4.1: User Interface

4.2 User interface

Although the user interface is not the primary scope of this thesis, we will swiftly introduce the reader to building UI in PureScript – the user interface is the ubiquitous component of web applications and also a reason why use Web Workers. The notion of interaction with user interface will accompany us for the most of the thesis.

For the sake of simplicity we have decided to use `purescript-jquery` [10] library for most of our examples – it consists of low-level `FFI` bindings to JQuery JavaScript library [11].

Apart from a few specific functions, we can omit most of the functionality offered by `purescript-jquery` without worries – we will focus only on functionality that enables us to:

1. Construct HTML elements and manipulate DOM (Document Object Model).
2. Handle user initiated events.

For the first task, constructing HTML elements, the library provides four important functions. We will start with introducing their type signatures [10]:

```
body :: forall eff. Eff (dom :: DOM | eff) JQuery
create :: forall eff. String
          -> Eff (dom :: DOM | eff) JQuery
setText :: forall eff. String -> JQuery
          -> Eff (dom :: DOM | eff) Unit
append :: forall eff. JQuery -> JQuery
          -> Eff (dom :: DOM | eff) Unit
```

Figure 4.2: Type signatures of remarkable functions in `purescript-jquery`

As you can see from the last part of type signatures, we are operating in context of `Eff` monad, with use of "extensible effects" explained in subsection 2.2.1, *Type system*. Although type signatures give us a hint of semantics of each individual function, we will describe their responsibilities explicitly:

`body` monadic action responsible for fetching root `<body>` DOM element of the current page.

`create` monadic action takes a parameter of type `String` carrying HTML tag. The action produces a value of type `JQuery`, which represents DOM element specified by the first parameter.

`setText` action takes a `String` and a value of type `JQuery`. The action is responsible for setting textual content of the DOM element passed as the second parameter.

`append` action is similar to `setText` – it appends the first DOM element to the list of children of the second DOM element.

Given the knowledge of functions described above, we can take a look at an actual usage in figure 4.3. The goal is to create a new heading with text "Hello World!:

```
main :: forall e. Eff (dom :: DOM | e) Unit
main = do
  h1   <- create "<h1>"
  setText "Hello World!" h1

  body <- body
  append h1 body
```

Figure 4.3: Constructing HTML elements in `purescript-jquery`

First we create new `<h1>` DOM element and set its content to "Hello World!" via `create` and `setText` functions.

Then we acquire page's root element, `<body>`, into which we insert our heading `<h1>` as a new child via the `append` function.

4.2.1 Handling user originated events

With user interface constructed in the previous section, we will now elaborate on adding interactivity to the UI. For this purpose the library `purescript-jquery` provides the following function:

```
on :: forall eff a. String
    -> (JQueryEvent -> JQuery -> Eff (dom :: DOM | eff) a)
    -> JQuery
    -> Eff (dom :: DOM | eff) Unit
```

The `on` function takes a name of an event, e.g. `"click"`, a function with side-effects that is triggered when the event happens (event handler), and a `JQuery` value representing the DOM element that will listen for this event.

Event handler is a function (of type signature being the same as the second argument of the `on` function), that takes `JQueryEvent` event data, a target of the event as a `JQuery` DOM element, and produces an effectful `Eff` monadic action.

Continuing with the example from figure 4.3, we will add *click* event handler on `<h1>` element, that will change text of the heading.

We bind the *click* event on the `h1` element to the event handler – function `handleClick` ignores event data and only changes the text of the target element. This can be seen in the example 4.4.

```
main :: forall e. Eff (dom :: DOM | e) Unit
main = do
  h1   <- create "<h1>"
  -- code from previous example omitted
  on "click" handleClick h1
  where
    handleClick :: JQueryEvent -> JQuery
                -> Eff (dom :: DOM | eff)
    handleClick _ element = setText "Web Workers" element
```

Figure 4.4: Handling events in `purescript-jquery`

Having covered both the HTML construction and the event handling, we can finally take a look at the first solution of the assignment.

4.3 Solution without Web Workers

In this section we will take a look at a simple solution of the assignment 4.1. The solution intentionally does not use Web Workers to outline the problem, which Web Workers are supposed to resolve – low responsiveness or freezing of user interface caused by CPU-heavy computation running in the main thread.

Also this solution serves to set a bar on simplicity (as mentioned in section 6.2, *Requirements and specification*) – we would like to reduce the Web Worker boilerplate code as much as possible, ideally to achieve a code very similar to the one without Web Workers.

4.3.1 Assignment implementation

As mentioned earlier the first solution of the assignment does not involve Web Workers – we will treat PBKDF2 as an ordinary computation running in the main thread. Doing so enables us to examine the underlying problem, explain its root causes and finally design a solution.

Regarding the implementation – while construction of HTML remains practically the same as in the UI introduction above, i.e. creating `<h1>` heading, `<button>` for interaction, and `<p>` element for displaying results, the implementation differs in the event handling as shown in figure 4.5.

```
handleClick :: forall f. JQuery -> JQuery
            -> JQueryEvent -> JQuery
            -> Eff (dom :: DOM | f) Unit
handleClick resultEl counterEl _ _ = do
  counter <- getText counterEl
  setText "Computing..." resultEl
  let key = pbkdf2 counter "blocking" 10000
  setText key resultEl
```

Figure 4.5: Event handler

`handleClick` event handler in 4.5, is responsible for fetching the counter value, computing the key using `pkbdf2` function, and showing the result in the UI.

`pbkdf2` function computing *Password-based Key Derivation Function 2* takes parameters standing for *password*, which is the value of the counter in our case, *cryptographic salt*, and *number of iterations*, which greatly influences the CPU time required to compute a key and duration of computation.

From the type perspective, `handleClick` function takes two `JQuery` values, one standing for DOM element displaying results and the other for the counter element. The rest of type signature should be familiar from `on` functions from `purescript-jquery`. Therefore partial application of `handleClick` is required in order to use the handler to bind to an event via `on`.

```
main :: forall e. Eff (dom :: DOM, timer :: TIMER | e) Unit
main = do
  -- redacted
  h1 <- create "<h1>"
  p <- create "<p>"
  button <- create "<button>"
  -- redacted

  runCounter h1 0
  on "click" (handleClick p h1) button
```

Figure 4.6: Fragment of `main` function with the `handleClick` function

The type signature of `main` function in the code 4.3.1 is worth noting as *extensible effects* (see 2.2.1) appear again in the form of `dom :: DOM`, which we have seen earlier, and newly as `timer :: TIMER`[2].

The `timer` effect is present because of `runCounter` monadic effect being invoked in the last part of the `main` function. This particular function is responsible for periodically updating the counter located in `<h1>` element.

```
runCounter :: forall f. JQuery -> Int
    -> Eff (dom :: DOM, timer :: TIMER | f) Unit
runCounter el cnt = void $ setTimeout 100 $ do
  setText (show cnt) el
  runCounter el (cnt + 1)
```

Figure 4.7: The counter responsible for periodical increments

The actual implementation of `runCounter` basically schedules a function to be executed after 100 ms pass – the function updates text of counter element, `el`, and recursively calls `runCounter` with incremented value. Recursive invocation then again schedules a counter update after another 100 ms.

2. The `TIMER` side effect is not related to measuring time, but represents JavaScript function responsible for scheduling repeated execution of a single function.

The full source code is available in the attachment in directory `demo/src/Demo/Blocking` and `demo/src/Worker/JavaScript`.

4.4 Flaws of the solution

This section examines the behaviour of the implementation introduced in the previous section and explains its flaws including root causes.

> To run the example either visit the online demo on `http://jandupal.com/web-workers-demo/` or see the instructions in `demo/src/README.md` to start a local web server.

Upon opening the HTML page in web browser we are presented with a choice of implementation. Selecting "#0 Blocking demo" option should display user interface as described earlier in this chapter – periodically incrementing counter and a button.

4.4.1 Blocking the main thread

When the button *Blocking Compute* is activated, the counter freezes for a noticeable amount of time. After a few seconds a generated key should appear under the counter and the counter should start counting upwards again. The period of time, during which the counter is not being updated, is a symptom of executing long-running computation in the main thread of browser's JavaScript engine.

Explanation of this phenomenon requires an analysis of the code execution from the perspective of JavaScript runtime *event loop*, described in section 2.1.2. Each execution of `runCounter` adds a message to the *event loop queue* marked to be executed *not sooner* then after 100 ms.

Clicking "Blocking Compute" button also adds a message to the queue – a message with the click event data and the assigned callback to be executed when processing the message.

As the JavaScript runtime model specifies, no interruptions or preemption takes place – in practice this means that no other JavaScript task is executed until the current task in the queue is processed completely. Moreover, not only execution of code is blocked until the pend-

ing task is completed, but also the user interface stops responding, due to the way browsers are implemented.

4.4.2 Web Worker technology comes to rescue

As shown above, any long-running task in the main thread reduces the responsiveness of any application. *Web Worker* technology offers a solution to mitigate this particular problem.

In the following chapters we will explore the Web Worker abilities and interface, set goals for PureScript integration and take a deeper look at proposed approaches.

5 Web Workers

Web Worker technology is defined by two standards - W3C Web Workers[3] and WHATWG Web Workers[4]. The specifications slightly differ in details as noted below, but both define Web Workers as:

> [...]API for running scripts in the background independently of any user interface scripts.
> This allows for long-running scripts that are not interrupted by scripts that respond to clicks or other user interactions, and allows long tasks to be executed without yielding to keep the page responsive.

Key points in the definition are *independence on user interface scripts* and *focus on long-running scripts*, which are both aspects examined in our demonstration assignment in chapter 4.

The specifications also mention that the workers are considered to be heavy-weight, thus should be spawned in smaller numbers and are expected to be long-lived in order to amortize high startup and per-instance costs [3, 4].

An equally important feature is the approach to communication between concurrent scripts. As opposed to shared-memory model, Web Workers do not allow sharing state between multiple scripts; *message-passing concurrency model* is offered instead [3, 4]. From this perspective, the workers can be seen as independent processes using sockets to communicate, rather than threads sharing memory.

5.1 Web Worker JavaScript API

There are two distinct types of workers available [3, 4]:

Dedicated Workers are linked to their creator script (e.g. browser tab) and only scripts running from the context (e.g. other workers) of the creator are able to communicate with the worker while

Shared Workers are named and any script being executed from the same origin (i.e. URL address) can obtain a reference and communicate with the worker.

For the scope of this work we will focus only on *Dedicated Workers* (further referred to as *workers*) – shared workers can be perceived as an extension of dedicated type. From this perspective *dedicated* type is the essential component that offers additional expressivity.

5.1.1 Spawning a new Web Worker

Dedicated Web Worker instances are represented by `Worker` interface, which defines constructor signature by W3C[1] [3]:

```
Worker(DOMString scriptURL)
```

The parameter `scriptURL` of type `DOMString` by W3C, respectively `USVString` by WHATWG – mapped to type `String` [12] – carries URL of JavaScript script to be executed in worker, e.g:

```
var worker = new Worker("/js/prime-numbers-worker.js");
```

This particular design decision, requiring a standalone JavaScript script for spawning a worker, greatly influences integration with PureScript as covered in following chapters.

When a worker is spawned, a separate *execution context* is created, backed by a thread, a standalone process, or similar construct. Such execution context includes an *event loop* (further described in 2.1.2) independent on its creator. [3, 4]

5.1.2 Message passing

To convey information from worker's creator to an actual worker, interface `Worker` offers means of *asynchronous message passing* [3, 4]:

`postMessage(any message)` a method which serves to send messages from creator to worker. Invoking the method adds new message to worker's *event loop*, which is later asynchronously processed by worker.

1. WHATWG specification [4] adds an optional second parameter `options`, which is relevant to ECMAScript 6 modules, which are not supported in PureScript as of December 2016.

`EventHandler onmessage` an attribute specifying event handler for messages coming from worker to creator. Such function is executed when a message from worker is being processed by creator's *event loop*.

Symmetrically, to enable worker to receive messages and send messages back to its creator, interface `DedicatedWorkerGlobalScope` accessible via `self` "inside" of worker defines a similar aspects [3, 4]:

`postMessage(any message)` a method for sending messages from the worker to the creator – processed by creators *event loop*.

`EventHandler onmessage` an event handler for receiving messages from the creator to the worker – processed by workers's *event loop*.

The key aspects are symmetricity of creator and worker APIs, *asynchronicity* of message passing and integration with receiver's *event loop*.

5.1.3 Termination

Worker can be terminated in two ways – from creator by invoking `terminate()` method defined by `Worker` interface, or from worker by invoking `close()` method defined by `WorkerGlobalScope`. [3, 4]

6 Implementing Web Workers in PureScript

This chapter opens the second part of the thesis: core work focused on extending PureScript ecosystem to support background execution, independent of the main script – integrating *Web Workers* JavaScript technology to PureScript language and tooling.

Before jumping straight to implementation details we first set goals and outline path to achieve them.

6.1 Design principles

When designing and implementing PureScript integration with Web Workers, we will primarily focus on two aspects: *simplicity* and *extensibility*. These two aspects are:

Simplicity, both in terms of comprehension complexity and implementation complexity. Both perspectives should be important in order to drive adoption of the library among PureScript community and support maintainability of the library.

Extensibility in sense that minimal restrictions are imposed when building higher abstractions based on top of our abstraction.

Although these principles are not formally specified and hard to evaluate objectively, they should serve us as a high-level guidance and help our intuition when designing and implementing solution.

6.2 Requirements and specification

This section describes requirements on different aspects of the solution. The requirements listed here will serve as a basis for evaluation of proposals in upcoming chapter 6, *Implementing Web Workers in PureScript*.

Acceptance criteria will be evaluated using the following grading:

Compliant – criteria met fully.

Needs improvement – criteria met partially, but not sufficiently.

Insufficient – criteria not met.

6.2.1 Type safety

As type safety is one of the strongly advertised advantages of PureScript, the solution is expected to provide type safety at level comparable with other PureScript libraries.

Main focus of this requirement is message passing – content, thus data type, of messages coveyed between creator and worker is defined by user. The library is expected to provide compile-time guarantees that such communication between creator and worker is always in compliance with specified data types.

Acceptance criteria The type checker fails during compile-time should the user attempt to send or receive incompatible type of message. Vice versa the type checker succeeds if type requirements are satisfied.

6.2.2 Matching PureScript ecosystem

In order to support adoption of the library the among PureScript community and ease the learning curve, the library is expected to use concepts familiar from the PureScript ecosystem and integrate well with existing PureScript libraries.

Acceptance criteria The library can be easily used in combination with selected PureScript user interface libraries.

6.2.3 Low tooling overhead

Tooling is an important component of software development process, PureScript is no exception – one of the popular build tools is *pulp*[13], which we will use to evaluate this aspect.

Acceptance criteria The library should not pose excess requirements on tooling. No intervention to the development process should be required and integrate well with the existing tooling.

6.2.4 Debugging

We expect the applications using Web Workers not to be exceptions in terms of occurrence of bugs when compared to other components of PureScript application. Thus the library should provide the same level of debugging experience as the rest of PureScript ecosystem.

Acceptance criteria The code executed in worker execution context can be debugged in the same way as the rest of the application.

6.3 Roadmap

We have chosen iterative approach to implementing Web Workers into PureScript – the progression starts with the least evolved solution and attempts to incrementally improve the deliverable:

1. Ad-hoc approach, covered in chapter 7
2. Library based on `Eff` monad, described in chapter 8
3. Asynchronous library based on `Aff` monad, depicted in chapter 9
4. Compiler-integrated, `forkIO`-like approach, detailed in chapter 10

In order to be able to better understand the proposals and objectively evaluate them, we will follow the following schema for each variant:

1. describe the proposal,
2. describe library functions, if any,
3. walkthrough example assignment defined in section 4.1,
4. evaluate fullfilment of requirements described in chapter 6.2

As a side topic, the following sections will also lead us through various problems, that were encountered while implementing the solutions – apart from investigating such problems, we will take a look at resolving them.

7 PureScript ad-hoc approach

When facing an implementation problem in PureScript, which requires use of native JavaScript API, *Foreign Function Interface* (described in section 2.2.2) allows us to explicitly separate specific parts of the code from the rest and express these parts directly in JavaScript – thus circumventing the fact, that a particular native JavaScript API is not fully exposed to PureScript. We have to provide entry points for executing JavaScript portions from PureScript context and vice versa – which is precisely use case for *FFI*.

In this chapter we will implement worker logic in a separate JavaScript file and provide means, possibly with type safety and explicit side-effects, to spawn a worker and send and receive messages.

7.1 Implementation

The implementation builds on the code described in section 4.3, *Solution without Web Workers*. The user interface is already in place, including event handlers – we focus on refining the *click* event handler to compute PBKDF2 key in a worker rather than directly in the handler.

7.1.1 Event handler

We will start with the `handleClick` event handler depicted in figure 7.1:

```
handleClick :: forall f. JQuery -> JQuery -> JQueryEvent
            -> JQuery -> Eff (dom :: DOM | f) Unit
handleClick resultEl counterEl _ _ = do
  counter <- getText counterEl
  setText "Computing..." resultEl
  computeKey counter "blocking" 10000 unsafePerformEff h
  where
    h = (flip setText resultEl)
```

Figure 7.1: Event handler

Note the difference in figure 7.1 when compared to our "solution without workers" in figure 4.5. The main difference is *inversion of control*, where the original solution itself displays resultant key via `setText`, while the current code invokes `computeHash`, which we will cover shortly, with partially evaluated `setText` monadic action as a parameter.

The concept of *inversion of control* allows callee to decide if and how will be the passed action executed. In our case this approach enables the code to defer the action (displaying result in user interface) until the worker finished computation, without blocking executing of the main script.

Next step is to examine implementation of `computeKey` function.

7.1.2 Worker invocation

Function `computeHash` serves as entry point to PBKDF2 computation in a worker. As usually we start with type signature:

```
computeHash :: forall e. String -> String -> Int
  -> (Eff e Unit -> Unit) -> (String -> Eff e Unit)
  -> Eff e Unit
```

The first three arguments are PBKDF2-related – first is *input* or *password*, second is *cryptographic salt* and the third is *number of iterations* as described earlier in section 4.1, *Assignment*.

The next argument of type `(Eff e Unit -> Unit)` may seem inappropriate at first glance, but looking back at the event handler implementation in figure 7.1, where the parameter is bound to the function `unsafePerformEff` [14] may give a hint. The parameter represents an *evaluator* of monadic `Eff` action passed as the last argument.

The last argument, `(String -> Eff e Unit)`, represents monadic action that takes place when worker finishes computation. The action is expected to carry out a side-effect, i.e. updating user interface to display result of computation. Notice that the result type of the action, `Eff e Unit`, matches type of *evaluator* passed as the fourth argument.

Regarding implementation of `computeHash` – the function is implemented via *Foreign Function Interface* (see 2.2.2), which in practice means PureScript file only defines its type signature, while the imple-

mentation is fully in JavaScript. The JavaScript implementation itself is responsible for:

1. Starting up a worker via `new Worker()`.
2. Setting up `onmessage` handler, which consists of evaluating monadic action – as outlined earlier.
3. Sending a message to the worker with PBKDF2 parameters.

The JavaScript part contains two noteworthy aspects – the first being call to `new Worker()`, which requires URL of worker script as a parameter. Such requirement implies a modification of tooling is required – details covered in following section 7.2.

The second noteworthy aspect is evaluation of the `Eff` monad in `onmessage` event handler:

```
worker.onmessage = function(event) {
  performEff(action(event.data));
};
```

The `action` argument is the `(String -> Eff e Unit)` argument – function responsible for processing message from worker. After applying the action to content of incoming message, we get a value of type `Eff` – monadic action. However the action is not evaluated yet – intuitively the value only describes "what is supposed to happen".

To carry out side-effects we need to evaluate the monadic action. For this purpose we use the parameter `(Eff e Unit -> Unit)` represented by argument `performEff`.

For full implementation, please refer to attachment – files `src/FFI/UI.purs` and `src/FF/UI.js`.

7.1.3 Worker implementation

As outlined in the beginning, actual worker is implemented in JavaScript. Key point here is the worker script is executed in a standalone execution context, thus it the script has to contain all dependencies, i.e. library for computing PBKDF2 in our case.

CommonJS [**common-js**] API, responsible for managing JavaScript modules, facilitates import of PBKDF2 functions to the worker script.

```
var pbkdf2 = require('pbkdf2');
```

Apart from the `require` statement, the script uses standard Web Worker API described in chapter 5, *Web Workers*, to receive messages from the user interface script and to send back messages containing resultant PBKDF2 key.

Full implementation can be found in the attachment – file `src/Worker/JavaScript/FF-Worke`

7.2 Problems

As can be seen, a large part of the solution is implemented in JavaScript. Although this approach is feasible and relatively little demanding in terms of development effort, it has other implications on quality and usability of the result. These implications are discussed bellow.

7.2.1 Lack of static typing

The large portion of JavaScript code has no *static typing* by its nature, which means we cannot benefit from PureScript type system providing us compile-time guarantees.

The weakest point in terms of type safety of the solution is located in message handling. Logic responsible for sending messages from user interface script and its counter part receiving messages in worker script are disconnected. Ideally these two distinct parts should be tied together by a type system to guarantee that type both sides expect the very same type of messages.

7.2.2 Build process and deployment overhead

As noted before, worker constructor `Worker(String scriptURL)` requires a single parameter representing URL of worker script.

This requirement directly clashes with PureScript build tool *pulp* [13]. In standard setup the tool compiles the whole PureScript application into a single compact JavaScript file, including dependencies. In order to accommodate the requirement, build process configuration has to be modified to generate separate worker script with its dependencies resolved.

Also this approach implies that the separate worker JavaScript file has to be uploaded and deployed on web server together with the rest of the application.

7.2.3 Flexibility

Obviously the approach with ad hoc worker integration may not scale very well in term of overhead related to development and build system – every time a new worker script is to be developed, new FFI mapping has to be implemented and a separate worker script created, including build system configuration.

7.3 Evaluation

From the practical point of view, such approach is feasible for simple ad hoc application of workers. However, a large-scale PureScript application may require a more integrated approach – as discussed in the following sections.

Evaluation of acceptance criteria defined in section 6.2, *Requirements and specification*, follows:

Type safety rated as **needs improvement**. PureScript's type system guards types of messages sent from PureScript to worker, but does not cover the other direction.

PureScript integration graded as **insufficient**. Although FFI is often used in libraries, the ad hoc approach used is not suitable to be published as a library.

Tooling overhead rated as **insufficient**. The approach requires to modify build system configuration and deployment too.

Debugging rated as **compliant**. Debugger tool available in modern browsers is able to set breakpoints and pause execution of worker scripts without any restrictions.

8 Proposal 1: Library based on `Eff` monad

The previous chapter listed deficiencies of a simple ad-hoc implementation of Web Workers technology in PureScript, which we will attempt to improve in this chapter – namely improving *type safety* and avoiding build process modification.

8.1 Build process integration

First we will cover steps to improve build process, because the approach influences design of type system described later in this chapter.

8.1.1 PureScript modules

The previous 7, introduced approach, where logic employed in *creator execution context* is separated from *worker execution context* logic. Improved approach depicted here keeps this concept, but on a different level of abstraction.

Previously the logic was separated on the level of compiled JavaScript files – one for creator and one for worker. The current approach separates logic on the level of **PureScript modules**.

Similarly to `Main` module, which usually serves as entry point to execution, each worker has a separate PureScript module. Such separation allows to benefit from standard compiler infrastructure such as dependency imports and provides natural encapsulation. Analogously to `main` function, worker modules have an entry point too – by our convention, functions executed at the beginning of worker lifecycle is called `default`.

An sample implementation of a worker module, which immediately after its initialization writes a message to browser console, is shown in figure 8.1.

8.1.2 Worker script URL

With a worker encapsulated in a module, we attempt to address another deficiency of the previous solution – having to compile and deploy two JavaScript files.

```
module ConsoleWorker where

-- imports omitted

default :: Eff (console :: CONSOLE) Unit
default = log "Worker started"
```

Figure 8.1: Worker encapsulated in a module

To recall why such situation occurred – JavaScript constructor of `Worker` requires a single parameter, which represents URL where the worker script is located (see chapter 5 for details). In the previous solution, we had to deploy both user interface and worker JavaScript files on a web server.

With worker logic extracted to a standalone module we could use the same approach, yet it would still require modification of build configuration.

Fortunately JavaScript has introduced *Blob objects* – an object that can be treated as a virtual file. Such object is created on runtime in browser, including its content. As opposed to file, blob objects are not persistent – they only live in browser memory and are discarded once JavaScript execution context ceases to exist. [**blob**]

Most importantly, an instance of `Blob` can be referenced by browser-local URL. Such URL is only temporary and accessible only from current execution context. However despite the limitations, we can easily use such URL to instantiate a worker.

General idea is to construct worker JavaScript source on runtime, create a `Blob` instance with the source and spawn a worker using the blob object.

8.1.3 Webworkify

Although constructing worker JavaScript source on runtime may not seem as a reasonable idea at first glance. However, a build tool `webworkify`[15] is able to preprocess the source file during compilation, so the worker script can be easily composed on runtime.

For proper function, `webworkify` requires the name of worker module to by statically specified in a `require` statement.

Thanks to this tool, we are able to create a blob object with a small effort, but gaining a reasonable improvement in terms of build system configuration.

8.2 Improving type safety

The primary problem of the previous solution in terms of type safety was a monolithic block of JavaScript code that was responsible both for messaging and PBKDF2 computation. As JavaScript is not a strictly typed language, type errors occur in run-time.

In efforts to address monolithicity via modularity we need to introduce types that carry information about messages transported back and forth between creator and worker. The ultimate goal is to forbid cases when one side send a message on one type while the receiving side expects a different type.

Note that as we have two distinct execution context – main script and worker, both with their own messaging API and application code – we have to bind these two context with a single type.

8.2.1 Message types

Communication between *user interface* and *worker* context can be seen as a sequence of **request** and **response** messages, with directions as depicted on figure 8.2. Although this terminology does not exactly cover all use cases (use case with responses but no requests is valid too), we will use the terminology for simpler reasoning.

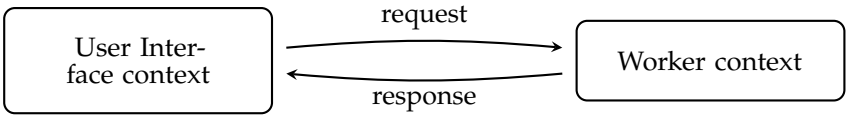

Figure 8.2: Communication between UI and worker contexts

To setup a type framework for our library, we start with reasoning about messages. Taking the PBKDF2 assignment as an example, *request*

type is represented by PBKDF2 parameters, say `KeyParams`. While *response type* is a `String`, which carries resultant key.

Sum type, or *tagged union*, can be used for cases, that require multiple types per direction. An example can be taken from HTTP protocol, when a request can have multiple types – GET, POST, UPDATE and other [16]. Multiple types can be unified into a single *sum type* in PureScript as follows:

```
data Request = GET String | POST Int | UPDATE Unit
```

As shown, for a single pair of creator and worker, two types suffice. Generally speaking the two types, *request type* and *response type*, do not have to be tied together as both may exists independently.

8.2.2 Worker types

The entity that ties *request* and *response* types is the actual worker, that expects incoming *requests* and sends back *responses*.

Before we define type for workers, let's take a look at cases where a type is useful:

Spawning a new worker to identify which worker script to spawn.

Passing worker identifier to other functions, similarly to passing `Worker` instance in JavaScript.

Sending message to guarantee type of message matches type of *request* or *response* required by worker definition.

Defining message handler to guarantee type of *request* matches the type of *message handler* and vice versa with *response*.

A different perspective, which influences design of types, is execution context:

Creator context usually user interface, which is responsible for *spawning worker*, sending *requests* and handling *responses* – corresponds to `Worker` JavaScript interface (see chapter 5).

Worker context which handles *requests* and sends back *responses* – analogous to `DedicatedWorkerGlobalScope` JavaScript interface.

To cover the space defined by these two dimensions, we introduce two distinct types:

`WorkerModule a b` represents PureScript module, which defines the worker logic – as outlined in previous section 8.1. The type is useful for spawning worker *creator execution context* and messaging API in *worker execution context*.

`WorkerId a b` serves to identify running worker instance. It is used for messaging API in *creator execution context*.

Note the type variables `a` and `b`, which specify types of *request* and *response*. The type variables provide *parametric polymorphism* – allowing us to preserve type information about messages and type-check messaging API on granular manner. Examples are shown in the next section.

8.3 Library

The library is divided into two modules by execution context – functions useful for *creator execution context* are located in `Master` module, while *worker execution context* functions reside in `Slave` module.

To demonstrate improved type safety, we will examine the `Master`, focusing on two aspects – type-checking **worker spawning** and **messaging** functions.

8.3.1 Type-checking worker spawn

As mentioned in previous sections, we have two types – `WorkerModule` and `WorkerId`. The former refers to worker script, while the latter to worker instance.

In order to convey information regarding message types from `WorkerModule` to `WorkerId` when spawning a worker, we introduce bind type variables of these two types in type signature of functions `startWorker`:

```
startWorker :: forall a b e. WorkerModule a b
            -> Eff (worker :: WORKER | e) (WorkerId a b)
```

Note type variables `a` and `b` bind type parameters of `WorkerModule` with `WorkerId`. A similar construct assists type-checking messaging functions.

8.3.2 Type-checking messaging functions

To ensure a message of proper type is send to a given worker instance, we need to match type of worker (its *request* type, precisely) with type of the message. Type signature of `sendMessage` function restricts this condition again by type variables:

```
sendMessage :: forall a b e. WorkerId a b -> a
            -> Eff (worker :: WORKER | e) Unit
```

Complementary function, `onMessage`, defining *response* message handling involves the very same mechanism for matching incoming message type with worker type.

8.4 Assignment implementation

The source code for this solution can be found in `demo/src/Demo/Eff/UI.purs`.

With types and API defined in the previous sections, we can finally showcase the approach via implementing the demonstration assignment. First we will outline structure of application, relations between components and their responsibilities.

The figure 8.3 sketches PureScript modules divided by two dimensions to four groups. The horizontal dimension describes which *excution context* uses which module, while the vertical dimension marks division of responsibilities between the assignment and the library.

The arrows depicted on the figure outline communication lines – `UI` starts a worker via `Master` module. The `Master` module takes care of spawning actual JavaScript worker and setting up message handlers. Analogously, the `KeyWorker` module sets up message handlers via `Slave` mode, that pair with messaging API on the creator side, and computes key via functionality provided by `PBKDF2` module.

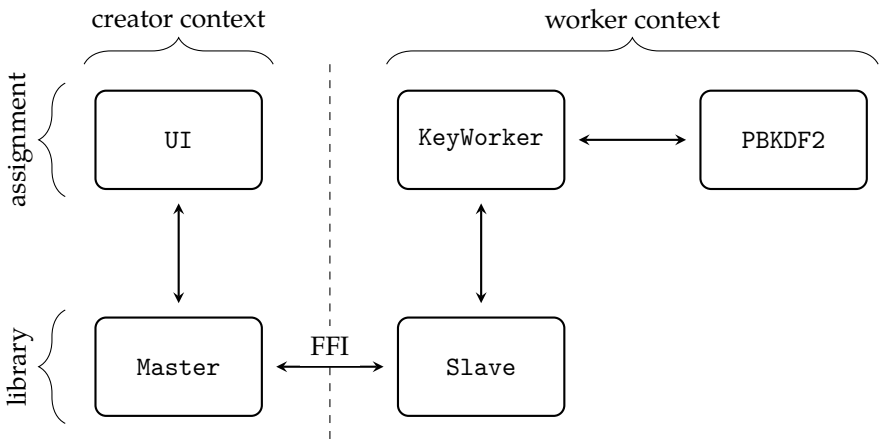

Figure 8.3: Components of assignment implementation

8.4.1 PBKDF2 Worker

As mentioned earlier in this chapter, worker logic is now isolated to a standalone PureScript module – named `HashWorker` in our assignment. The library requires the module to contain following:

`WorkerId` and `WorkerModule` definitions which both specify types of *request* and *response* messages.

`default` function defining procedure, which is executed in *worker context* immediately after it is spawned.

To satisfy the first item, `HashWorker` module defines two elements – `KeyWorkerId` type alias and `workerModule` value of type `WorkerModule`. The implementation of `workerModule` value requires a small usage of *foreign function interface* – as section 8.1.3 states, `webworkify` needs JavaScript `require()` with name of module, `"KeyWorker"` in our case, to work properly.

Therefore `workerModule` value is defined in `HashWorker.js` as follows:

```
exports.workerModule = require("KeyWorker");
```

Thanks to this single line in JavaScript, we do not have to modify build system settings to produce separate JavaScript files.

The `default` function is expected to setup message handlers for incoming messages from the main thread.

To achieve this, it uses `onMessage` library function available in `Control.Monad.Eff.Worker.Slave`. To send a message back the function `sendMessage` is available in the very same module:

```
default :: forall e. Eff (worker :: WORKER | e) Unit
default = onMessage workerModule handler
  where
    handler msg = sendMessage workerModule $ computeKey msg
```

This simple example of `default` function demonstrates handling of incoming message via localy-defined `handler` function, computing PBKDF2 key and sending the result back via `handler`.

8.4.2 Spawning worker and sending messages

With the library in place, spawning the worker and defining message handler in *creator execution context* is a matter of call to the functions `startWorker` and `onMessage`, which are exposed via library module `Control.Monad.Eff.Worker.Master`:

```
main :: forall e. Eff (dom :: DOM, timer :: TIMER, worker
↪  :: WORKER | e) Unit
main = do
  p <- create "<p>"
  -- redacted
  worker <- startWorker workerModule
  onMessage worker (\msg -> setText msg p)
```

Similarly as in the previous section, `onMessage` function handles incoming messages. However this time it does not compute PBKDF2 key, but updates the user interface via `setText` function as defined by the *lambda function*.

The act of sending message to worker happens within `handleClick` event handler, which was introduced back in 4.4. The relevant fragment of the `main` function is shown bellow:

```
computeHash :: forall f. KeyWorkerId -> String
            -> Eff (worker :: WORKER | f) Unit
computeHash worker input = sendMessage worker p
  where
    p = { input: input, salt: "sync", iterations: 10000 }
```

Basically the library function `sendMessage` is invoked with the proper arguments and the underlying library takes care of the rest.

The full source code can be in file `demo/src/Demo/Blocking/UI.purs`.

8.5 Problems

Although the approach described in this chapter resolved *type safety* and *build system integration* problems mentioned in chapter 7, *PureScript ad-hoc approach*, other problems appeared.

8.5.1 Debugging

Using *blob* object for constructing worker script on-the-fly resolved problem with *build system*, but it has introduced a debugging limitation – as a new blob object is created each time a worker is spawned, browser developer tool does not retain breakpoints in worker logic set up previously.

As a workaround, `debugger` statement can be used to force the browser to pause the execution.

8.5.2 Message handler is executed multiple times

While this is not a technical problem, we believe it is worth noting.

When reasoning about the `main` function in section 8.4.2, it may not be immediately clear that the *lambda function* message handler, `(\msg -> setText msg p)`, is actually executed multiple times – each time a message is received.

A concept of applying a given function on each element of a sequence of inputs is already well known in *functional programming* – it corresponds to `map` function or `mapM` function for the case of monads.

Explicitly modeling this fact in the library would improve the learning curve and enhance compatibility of the library with other asynchronous PureScript libraries.

8.6 Evaluation

> To run the example either visit the online demo on `http://jandupal.com/web-workers-demo/` or see the instructions in `demo/src/README.md` to start a local web server.

With *type safety* and *build system integration* from the previous chapter 7, *PureScript ad-hoc approach*, addressed we have significantly improved support for Web Workers in PureScript. Evaluation of acceptance criteria defined in section 6.2, *Requirements and specification*, follows:

Type safety rated as **compliant**. The library has improved type safety by introducing new types and enforcing their usage in library functions.

PureScript integration graded as **needs improvement**. Although `Eff` monad is a popular and wide-spread abstraction, worker messages are by their nature *asynchronous*. The library could employ PureScript's `Aff` monad to model *asynchronous* computations to improve semantics.

Tooling overhead rated as **compliant**. Overhead in the form of compiling multiple JavaScript files was successfully eliminated by using `webworkify`.

Debugging rated as **needs improvement**. Workaround described in section 8.5.1.

Two aspects remain partially resolved – *PureScript integration* and *debugging*. Our attempt to address the former is described in the following chapter.

9 Proposal 2: Asynchronous library based on `Aff` monad

One of the popular PureScript monads is `Aff` monad [17] focused on modeling *asynchronous* computations. The main reason to model asynchronous computation differently than by passing *handlers* (or *callbacks*), as we have done in our examples, is so called "callback hell".

The term "callback hell" refers to sequencing multiple asynchronous operations, in which the result of one operation is passed as input to the subsequent operation. Usual approach with *handlers* fails as the first handler needs to define a subsequent handler in its body. The more operations are sequenced the more nesting of handlers is required to achieve the specified goal. [17]

To address this issue, the `Aff` monad introduces a set of constructs that allows developers to treat an asynchronous computation as a synchronous one. A simple demonstration[17] of this feature follows:

```
main = launchAff do
  response <- Ajax.get "http://foo.bar"
  liftEff $ log response.body
```

The function `Ajax.get` is an asynchronous operation, which fetches the content of a given URL via HTTP. However instead of passing a *handler* to the function, we use `<-` (or *bind*) making the code looking like an ordinary synchronous imperative code. Yet under the hood the `Aff` monad still works with *handlers* to achieve asynchronicity.

While the example above may seem like a *syntactic sugar*, the `Aff` monad provides more functionality – e.g. the `AVar` [17], asynchronous variables similar to Haskell's `MVar` structures [8].

9.1 Modeling asynchronous computation

Using the `AVar` structure allows us to model a mutable state with *asynchronous read and write* operations – the asynchronicity means that an *asynchronous write* operation is not triggered (i.e. it "blocks" `Aff` computation) until an *asynchronous read* operation is requested. [17]

And vice versa *asynchronous read* operation will "block" asynchronous computation until a value is written to the variable via *asynchronous write* operation.

From a different perspective, the `AVar` is an *asynchronous queue*.

9.1.1 Modeling message passing as a queue

The desirable state of inbound message handling from the developer's point of view is outlined as follows. Both `forever` and `takeVar` functions are provided by the `Aff` monad [17].

```
main = forever $ do
         message <- takeVar queue
         updateUI message
```

Basically we explicitly state via the `forever` function, that the action of dequeuing the queue by `takeVar` and updating UI, will be executed indefinitely.

Normally in the `Eff` monad simlar `main` method would result in blocked main thread. However in the `Aff` monad there is no *active waiting* involved as popping the `queue` is internally handled asynchronously via handlers.

Sending a message is done similarly with the exception of `putVar` function from the `Aff` monad being used instead of `takeVar`.

9.2 Library

To reach the desired state described in the previous section, we need to expose a pair of queues represented by `AVar` values – one queue for *incoming* messages and one for *outgoing* messages.

9.2.1 Exposing queues of messages

To encapsulate worker messaging as a pair of queues, we introduce the function `makeChan` from module `Control.Monad.Aff.Worker.Master`:

```
makeChan :: forall req res e. WorkerId req res
         -> Aff e (Tuple (AVar req) (AVar res))
```

The `makeChan` function accepts an instance of running worker and returns a pair of queues with type corresponding to worker's *request* and *response* message types. Any message that is written to the first `AVar` is automatically send to the worker and any message received from the worker is automatically enqueued to the second queue.

As a pleasant byproduct, we've managed to *decouple* the messaging infrastructure from worker API. Any algorithm compatible with the `AVar` asynchronous variables will be able to utilize them, while being perfectly isolated from the fact that a web worker happens to be on the other side of the queues.

Internally we asynchronously *poll* for messages from *outgoing message queue*, which we send to the worker, respectively creator, by the `sendMessage` function from our `Eff` library.

To populate the *incoming message queue*, we register a *handler* via `onMessage` function introduced in the previous chapter. The handler basically enqueues every incoming message to the *incomming message queue*.

9.3 Assignment implementation

As we have transitioned from using the synchronous `Eff` monad to employing the asynchronous `Aff` monad in library, we need a similar transition on the side of the UI library.

In the previous sections we have been using the `purescript-jquery` `Eff` library, but the purpose of this chapter we need an `Aff` user interface library to fully demonstrate the solution.

9.3.1 Halogen UI

The *Halogen*[18] user interface library takes a diametrically different approach to user interface programming than the `purescript-jquery` library – it strictly separates *application state*, *UI rendering*, and *interactivity*.

Being built on top of the `Aff` monad, the library is a good choice for our demonstration.

For further information please refer to Halogen documentation [18].

9.3.2 Implementation

The source code for this solution can be found in `demo/src/Demo/Aff/UI.purs` file.

When inspecting the implementation provided in the attachment, note the way the worker is initialized. Although we use the same function, `startWork`, to spawn the worker, there are no other occurences of our `Eff` library. Right after spawning the worker, we use the `makeChan` function to build the queues and from that point we use only the queues.

Thanks to the decoupling, we can minimize interfaces – the application state is sufficient only storing the *outgoing message queue*, ignoring the fact that the *incoming message queue* exists, even that there is a web worker running.

Moreover, the logic responsible for processing *incoming message queue* is neatly isolated and only communicates with the application by emitting a *halogen action* whenever a message from worker arrives.

9.4 Evaluation

By providing the `Aff`-compatible interface for our library, we were able to simplify the library, improve integration of Web Workers into asynchronous computations and achieve a better decoupling by hiding the complexity behind two `AVar` queues.

Evaluation of acceptance criteria defined in section 6.2, *Requirements and specification*, follows:

Type safety rated as **compliant**. The level of type safety remains the same as in the case of our `Eff`-based library.

PureScript integration graded as **compliant**. Integration with PureScript ecosystem was improved by providing asynchronous interface and decoupling messaging from Web Workers.

Tooling overhead rated as **compliant**. No changes from the `Eff`-based library.

Debugging rated as **needs improvement**. Unfortunately the debugging issue is still present.

10 Future work: Compiler-integrated, `forkIO`-like approach

As opposed to the previous chapters, which described the actual implementation in `purescript-web-workers` library, this chapter proposes a topic for a future work – improving the Web Worker integration beyond the state of the art.

Inspiration for this topic is Haskell's `forkIO`[8] function, which takes an `IO` action and executes the action in a concurrent thread.

Having a similar construct in PureScript would allow to simplify our solution even more. The developer would not have to define a separate module for the worker logic, spawn the worker etc. A hypothetical example using the `AVar` variable follows:

```
main = do
  var <- makeVar
  forkIO $ putVar var (pbkdf2 "password")
  takeVar var >>= updateUI
```

The proposed implementation of PureScript `forkIO`-like function would be responsible for composing worker script, spawning the worker, and setting up messaging behind the scenes to deliver an illusion of Haskell's thread.

To our best knowledge, no functional language targeting JavaScript currently has such a seamless integration of Web Workers. PureScript language could benefit highly if implemented.

In order to fully implement the proposal the following items need to be resolved:

Dynamical generation of worker scripts The current library requires the worker scripts (modules) to be defined statically, in advance. Otherwise the approach with `webworkify`, described in section 8.1.3, would not work.

Serialization of function closures As the example shows, the `putVar` action's first parameter belongs to the scope of the `main` function. But it would have to be accessible in both *creator execution context* and *worker execution context*, thus the implementation would

have to be responsible for transporting such values back and forth between contexts. The *Cloud Haskell*[19] project addresses a similar case in a distributed environment.

Compiler extension In order to support the two previous items, the PureScript compiler would have to be modified to provide more information for libraries implementing the advance use-cases. Also a decision, whether such feature is available only in JavaScript target language or extended to other PureScript backends too, would have to be made.

Perfomance costs As stated by the Web Workers standards [3, 4], workers are considered heavy-weight in terms of startup costs and memory requirements. The ease of use of the `forkIO` function could lead to excessive usage, thus negatively affecting performance.

11 Conclusion

This thesis has briefly introduced the reader to the domain of the PureScript functional programming language in relation to JavaScript, the target language of the PureScript compiler. The problem statement, *concurrent execution of client-side scripts in PureScript*, is presented with an apparent demonstration to advocate the motivation and to deepen the intuition behind the problem rooted in the JavaScript *runtime model*.

The second part of this paper explores the *Web Workers* JavaScript standard and defines a roadmap to deliver a PureScript library to mitigate *the problem*. The three-step iterative development process of the library is described including design choices, encountered problems, and technical solutions. Each of the three steps is also concluded with a demonstration and evaluation. In the third step, the deliverable, `purescript-web-workers` library, is assessed as suitable for mitigating *the problem*, thus meeting the goal of this thesis.

Finally, in the last chapter we propose a future work to push the boundaries of Web Worker integration even further by overcoming the worker limitations.

Bibliography

[1] Dave Raggett. *Client-side Scripting and HTML*. W3C Working Draft. [Online; accessed 2-Jan-2017]. Mar. 1997.

[2] Phil Freeman. *PureScript by Example*. [Online; accessed 10-December-2016]. Leanpub, 2016. URL: https://leanpub.com/purescript.

[3] Ian Hickson. *Web Workers*. W3C Working Draft. [Online; accessed 4-December-2016]. Sept. 2015.

[4] WHATWG. *Web worker*. [Online; accessed 4-December-2016]. Dec. 2016. URL: https://html.spec.whatwg.org/multipage/workers.html.

[5] David Flanagan. *JavaScript: The definitive guide: Activate your web pages*. " O'Reilly Media, Inc.", 2011.

[6] Mozilla Developer Network. *Concurrency model and Event Loop*. [Online; accessed 2-January-2017]. 2013. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop.

[7] V Nazarov, H Mackenzie, and L Stegeman. "GHCJS Haskell to JavaScript Compiler". In: *Retrieved December* 1 (2016).

[8] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. "Concurrent haskell". In: *POPL*. Vol. 96. 1996, pp. 295–308.

[9] Burt Kaliski. "PKCS# 5: Password-based cryptography specification version 2.0". In: (2000).

[10] Phil Freeman. *purescript-jquery: Type declarations for jQuery*. https://pursuit.purescript.org/packages/purescript-eff/2.0.0. Version 2.0.0. [Online; accessed 20-December-2016]. Sept. 2016.

[11] Bear Bibeault and Yehuda Kats. *jQuery in Action*. Dreamtech Press, 2008.

[12] Tobie Langel Cameron McCormack Boris Zbarsky. *Web IDL*. W3C Editor's Draft. [Online; accessed 29-December-2016]. Dec. 2016.

[13] Bodil Stokke. *pulp: A build tool for PureScript projects*. https://github.com/bodil/pulp. Version 10.0.0. [Online; accessed 2-January-2017]. Nov. 2016.

[14] Gary Burgess. *purescript-eff: The Eff monad, for handling native side effects*. https://pursuit.purescript.org/packages/purescript-eff/2.0.0. Version 2.0.0. [Online; accessed 15-December-2016]. Sept. 2016.

[15] Anand Thakker. *webworkify: Launch a web worker that can require() in the browser with browserify*. `https://www.npmjs.com/package/webworkify`. Version 1.4.0. [Online; accessed 2-January-2017]. Dec. 2016.

[16] Roy Fielding et al. *Hypertext transfer protocol–HTTP/1.1*. Tech. rep. 1999.

[17] Gary Burgess. *purescript-aff: An asynchronous effect monad for PureScript*. `https://pursuit.purescript.org/packages/purescript-aff/2.0.2`. Version 2.0.2. [Online; accessed 26-December-2016]. Dec. 2016.

[18] Gary Burgess. *purescript-halogen: A declarative, type-safe UI library for PureScript.* `https://pursuit.purescript.org/packages/purescript-eff/0.12.0`. Version 0.12.0. [Online; accessed 22-December-2016]. Dec. 2016.

[19] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. "Towards Haskell in the cloud". In: *ACM SIGPLAN Notices*. Vol. 46. 12. ACM. 2011, pp. 118–129.

A Directories in attachment

The digital archive contains contains two directories:

purescript-web-workers containing the `purescript-web-workers` library implementation.

demo containing assignment implementations as incrementaly covered in the thesis.

To start a local web server to examine demonstration assignment implementations please see `demo/README.MD`.