University of Bergen
Department of Informatics

Flushable Promises: Modular Event Coalescing in Graphical User Interface Programming

Author: Maria Katrin Bonde
Supervisors: Knut Anders Stokke, Jaakko Timo Henrik Järvi, Mikhail Barash

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2024

Abstract

Graphical user interfaces (GUIs) are everywhere. They serve as the primary means of interaction between an application and its users. Developing GUIs with complex dataflows, where multiple elements depend on each other's input, is challenging. GUI dataflows often include asynchronous operations, and are typically implemented as chains—or graphs—of JavaScript promises.

Event coalescing strategies, like debouncing and throttling, are common techniques for dealing with a large number of GUI events. In GUIs with complex dataflows, these techniques are a source of subtle bugs: event coalescing easily breaks modularity. When a program needs to get data from a promise at the end of a dataflow, the programmer needs to know if the source of the data is being delayed by event coalescing.

This thesis proposes an extension to the JavaScript promise abstraction: the ability to "flush" a promise chain. An arbitrary promise anywhere in a promise chain can signal the first promise of the promise chain to resolve. We call this extended version of promises *flushable promises*. With flushable promises, event coalescing strategies can be implemented in a less fragile manner, improving the modularity and separation of concerns in programming GUIs with complex dataflows.

Acknowledgements

I would first like to express my gratitude to my supervisors for all their support: Knut Anders Stokke, for his immense help, sitting with me for hours on end and helping with every aspect of my thesis; Professor Jaakko Timo Henrik Järvi, for his words of encouragement and detailed feedback on my writing; and Associate Professor Mikhail Barash, for his pep talks and always being ready to answer any questions I might have.

Secondly, I want to thank my boyfriend Knut for all his care, reassurance and uplifting words when they were needed.

Finally, I would also like to thank my parents, my sisters and my friends for their support throughout my studies.

Maria Katrin Bonde
Tuesday 11th June, 2024

Contents

List of Figures

Listings

Chapter 1

Introduction

In modern software development, graphical user interfaces (GUIs) are inescapable in many applications. In such applications, the GUI acts as the main interface through which users interact with and operate the application. Developers and designers are thus continuously working on improving the intuitiveness, response time and overall user experience of their GUIs.

Different fields, widgets and visual elements in a GUI may be related to each other so that when users interact with one, the others are updated accordingly. The GUI's *dataflow* defines how data propagates from one widget to other widgets. It can be challenging for a programmer to create GUIs with complex dataflows that remain responsive and always behave as expected, especially if the program has dependencies both ways between several GUI variables, that is, networks of variables where data can flow in multiple directions [45].

Multi-way dataflow constraint system [31] programming enables developing GUIs with complex dataflows, by establishing the relations between variables as constraint-enforcing methods. *HotDrink* [33] is a JavaScript library that implements multi-way dataflow constraint systems by representing every variable as a *promise*. Promises (or sometimes "futures") are many programming languages' default way to represent asynchronous operations and can be chained to create sequential asynchronous operations, where one operation's execution depends on the preceding operation's completion. HotDrink employs promises to create networks of sequential operations on variables, where one promise's value depends on one or several other promises' values. This network forms a graph of dependencies, a *promise graph*.

The HotDrink library manages the dynamic dataflows of GUIs: during user interaction, new dataflows are computed based on the order in which the variables have been edited, and for each dataflow a new promise graph is constructed. When a programmer develops a GUI using HotDrink, they do not need to care about the order in which the variables are updated, or which sources are used to compute new values for the variables. This allows for a modular GUI and separation of concerns—the programmer should not have to worry about the order as long as the outcome is always correct.

The source of a variable update may stem from user interaction with the widgets. Whenever the values of these widgets change, the new data is propagated through a new promise graph. Some widgets will typically encounter rapid user events: a text input field may produce an event for every keystroke, and a slider may produce hundreds of events as users move the slider to a new value.

Every such event may not need processing. If many similar events occur in a short period of time, and especially if the event handler causes potentially computationally expensive or time-consuming network requests, the program may benefit from not processing every event. Additionally, if continuously occurring events cause rapid re-rendering of the GUI, the user experience may worsen as the page is constantly updated.

To not process every event we can employ *event coalescing strategies* [41], which are strategies to manage rapid events, to combine multiple events into one. Two such event coalescing strategies commonly used are *throttling* and *debouncing*. A throttled event-handling function will have a period after it executes in which all calls to the same function will be ignored. The execution of a debounced function will be delayed until it is not called for a fixed period of time. Both strategies are useful and can improve our user experience, but their obvious implementations are a source of subtle problems and can lead to error-prone GUIs.

In programs with complicated graphs of promises such as those maintained by HotDrink, these event coalescing strategies break modularity: The part of the program that uses data at the end of the dataflow may not be aware that event coalescing is happening at the sources. The essence of the problem is that when delays are implemented naively, the programmer must keep track of whether an asynchronous task is being delayed or not. This problem is prevalent in both small- and large-scale applications.

An example of this issue is a form for users to input their email, where the form validates whether it is an accepted email or not. To avoid validation checks at every typing event, the validation checks are debounced. If users type quickly and then press

Enter or outside the input field, a validation check should be executed immediately. The variable containing the email input may not have been updated with the latest keystroke events, and therefore the pressing of Enter triggers a validation check on a different string than what the user actually typed. To rectify this, the event handler for Enter would have to be aware that the validation check's data source was debounced, and also know how to update the variable with the correct value before triggering the check. Even for such a simple dependency graph, the modularity is broken, which can consequently be experienced from the many erroneous validations on website forms.

In this thesis, we propose extending promises with functionality to better handle event coalescing: Promises waiting on values that are delayed—due to event coalescing—have the ability to `flush`, that is, resolve with the correct data immediately if possible. Moreover, when using data from a promise at the end of a promise graph, programmers can, through this promise, trigger all its source promises to flush. We believe this improves programming with complicated dataflows where reasoning about how promises are related within a promise is non-trivial. We call this abstraction *flushable promises*.

With the help of flushable promises, event coalescing strategies can be implemented in a less brittle way: when processing data at the end of a dataflow, programmers do not need to worry about which widgets or input sources throttle, debounce, or do neither.

Chapter 2

Background

To program GUIs that have no unsurprising behavior while maintaining complex dataflows, we introduce an abstraction building upon promises in JavaScript. To understand how this abstraction works and its usefulness, this chapter gives a solid foundation of knowledge regarding both the root of the problem we are solving and the components that are used in our solution.

The material in this chapter is ordered as follows. First, we explain the event coalescing strategies debouncing and throttling and their respective applications, which is followed by an exposition of the concepts of asynchronous event handling. Here we follow the course of the history of asynchronous programming in JavaScript, which starts with callbacks. We subsequently explain JavaScript promises, an asynchronous datatype, which is then followed by generators and finally the abstraction `async/await`. Furthermore, we look at the complexity that arises when combining debouncing or throttling with promises. Lastly, we describe how multi-way dataflow constraint systems benefit end-users by making it easy for programmers to define complicated dataflows for their GUIs, and how the complexity of event coalescing strategies is intensified in constraint system-powered GUIs.

2.1 Event coalescing strategies

When implementing user interfaces, we must often consider the case that some user input may cause a lot of calls with the same or similar input to the same event-handling

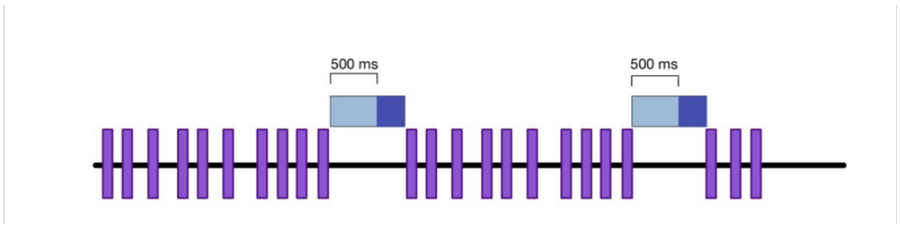


Figure 2.1: Illustration of calls to a debounced function.

function. In GUI implementations, there are event listeners and event-handling functions. The event listener fires every time an event happens, but we can limit the number of times the event-handling function is executed. This can be beneficial because if the function calls become too frequent, a program can slow down as we are overloading the program. The event coalescing strategies discussed in this chapter limit the amount of work by deciding when to trigger further processing of the event, when to ignore the event, and when to delay further processing to first see if more user input is happening. Thereby not executing the event-handling function every time it is called by the program.

2.1.1 Debouncing

Debouncing [36] is an event coalescing strategy where we include a brief delay between the moment a function is called and when it is executed. During this waiting period, if the function is called again, the pending execution is canceled and the waiting period is reset. Hence, the function will not be executed until a delay is concluded without another function call interruption. To further illustrate the concept of debouncing, in Figure 2.1, the purple bars represent events, and the light blue squares represent the compulsory delay. In this figure, the delay is set to half a second. The dark blue squares represent function execution, that is, where the delay is permitted to finish. The purpose of debouncing is to ensure that the function in question is not invoked with intermediate input, and in this way prevent unnecessary and excessive execution of function calls.

In certain GUI scenarios, debouncing is a much-used technique. For example, when a user types into a search bar and we want to show search suggestions, it is customary to debounce the input events a few hundred milliseconds as we do not want to send a new request on every keystroke. In most cases, the user will type the first few letters of the word they are after, and therefore we would want to wait until they pause their typing before sending the call to show them the search suggestions. Debouncing is a good technique for such situations.

In other cases, it is clear that debouncing is not the right choice of delay strategy. An example of this is a button in a GUI where the user typically expects an immediate outcome. In the case that a user presses a button, and nothing happens right away, the then frustrated user might rapidly press the same button a few more times. If the function handling the button press is debounced, the more the user presses the button, the more its action is delayed.

2.1.2 Throttling

Another technique for limiting event processing work is *throttling*. When an event-handling function is throttled, a waiting period will be included *after* the call to the processing function is executed, in which all the following events attempting to call said function are ignored. The first function call after the waiting period is over is once again executed. Figure 2.2 illustrates the calls to a throttled function; the function is invoked after the first event, and the events that follow are ignored. The first event that occurs after the five-second wait time is processed. Events that occur during the wait time will be ignored, as illustrated in the figure with the last three events. The events are processed regularly no matter the frequency of events.

Throttling is recommended for curbing large quantities of input events in a range of GUIs, for such tasks as resizing windows, scrolling, or typing that trigger fetching from APIs. No matter the frequency of the events, if the event handler is throttled, the events will be processed regularly. Hence, the user will not have to pause their user interaction to observe a change in the GUI. In use cases where the user might input a lot of events, but it is fine to ignore a few function calls in between executing them, throttling might be useful. The previously mentioned scenario of the frustrated user pressing excessively on the same button to hasten the event handling is an example of where throttling could be implemented to avoid unnecessary function calls. Since a button press event carries no other information than that the button was pressed, it is usually sufficient to process just the first event of a sequence of repeating events. If the button triggers a network request, and the user clicks the button several times while the application is waiting for the response, there is usually no point in sending the same network request many times. The throttled event handler for the button can process the first click and safely ignore any subsequent clicks for a period of time.

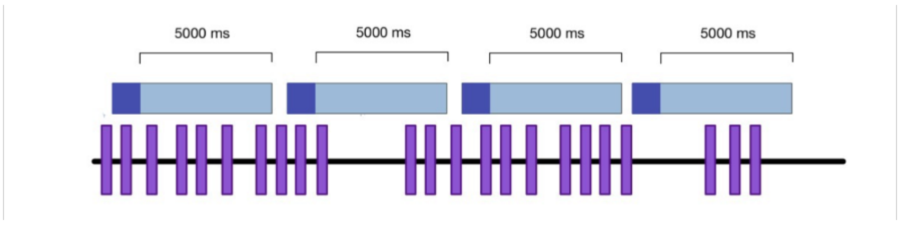


Figure 2.2: Illustration of calls to a throttled function.

2.1.3 Implementing debouncing and throttling in JavaScript

The `debounce` function implemented in Listing 2.1 takes the event handler function and the waiting period as parameters. The implementation schedules a function to be called, and if the function is called again within the given time frame, the previously scheduled function call must be stopped; the variable `scheduledCall` will store a reference to the currently scheduled call so that it can be unscheduled. The `debounce` function then returns another function that schedules the event handler function, which first stops the currently scheduled call, if any. Thus, if the debounce function is called before a timeout is over, that scheduled call is canceled and a new one begins, producing the behavior of debouncing discussed in the previous section. *Spread syntax* (e.g. `(...args)`) is used here to forward all the arguments to a call to `func`[1].

Listing 2.1: Debounce implementation.

```
function debounce(func, delay) {
  let scheduledCall = null;
  return (...args) => {
    clearTimeout(scheduledCall);
    scheduledCall = setTimeout(() => {
        func(...args);
    }, delay);
  };
}
```

The implementation of throttling is shown in Listing 2.2. The variable `acceptNextCall` decides whether a call to the throttled `func` can be executed or not. The first time `throttle` is called, `acceptNextCall` is `true`. To make sure there are no timers currently running, we check if the `acceptNextCall` variable is `true`. If that is the case, we immediately set the `acceptNextCall` to `false`, and we call the throttled function and set a

[1]The spread syntax in a function's parameter list accepts any number of arguments and in a function call passes the arguments on. Some implementations of debouncing are more simplistic and do not use the spread operator. Such implementations only work with handler functions that take no arguments.

timeout. Any calls made during the timeout will be ignored. After this timeout is over, the `acceptNextCall` is set to `true` again, and so any future event can again call `func`.

Listing 2.2: Throttling implementation.

```javascript
function throttle(func, delay) {
    let acceptNextCall = true;
    return (...args) => {
        if (acceptNextCall) {
            acceptNextCall = false;
            func(...args);
            setTimeout(() => {
                acceptNextCall = true;
            }, delay);
        }
    };
}
```

Note that both implementations return a function, and we refer to these functions as the *inner function bodies*. When employing these delay strategies, we do not call `debounce` or `throttle` directly every time an event happens. The correct way to employ these functions—in this case, debouncing—is illustrated in Listing 2.3. When `handleEvent` is initialized, the code outside the inner function body of `debounce` is executed. As mentioned `debounce` returns a function, which is stored in `handleEvent`. We add `handleEvent` as the handler of an event listener. Thus, `handleEvent` will be called every time an event happens to `myButton`, but inside `handleEvent`, `processValue` will only be executed if the delay finishes.

Listing 2.3: Use of debounce.

```javascript
const handleEvent = debounce(() => processValue(), 250);
myButton.addEventListener(() => handleEvent());
```

2.2 Asynchronous programming in JavaScript

Concurrency is an important aspect of GUI programming, as GUIs should remain responsive to user interaction while performing potentially time-consuming operations. Therefore, in addition to event coalescing strategies such as debouncing and throttling, asynchronous event-handling plays a pivotal role in enhancing an application's responsiveness and efficiency. These properties are especially important because they can often be the primary factors contributing to whether a GUI offers a good user experience or not. To understand how the different asynchronous mechanisms in JavaScript work, we need to know how JavaScript handles concurrency and chooses which code to execute.

JavaScript is a single-threaded language, with one thread called the "thread of execution" [29], or in the context of GUI programming, the "UI thread", as its purpose is to queue and respond to tasks created by user interaction. Every event will be added to the back of the *event dispatch queue* paired with the code expected to run when the UI thread picks it up. The UI thread picks the event at the front of the queue and runs the code paired with the event [29]. When the UI thread is handling an event, the page it is running on is frozen. If a task takes a few milliseconds to execute, this is not a problem, as the user will most likely not notice. However, when performing tasks such as retrieving large amounts of data from a server, keeping the UI thread waiting for the response could potentially freeze the page for a noticeable amount of time. This is the reason we need asynchronous event handling.

2.2.1 The JavaScript event loop

The JavaScript event loop determines how events are handled by an application. The event loop checks the front of the event queue and matches the first event to an event-handler [55, Ch.2]. The event queue is (at least) two separate queues, which hold tasks performed by the browser. The queues are sometimes referred to as macroqueue and microqueue, as they make a distinction between microtasks and macrotasks; a macrotask can be parsing of HTML code, altering the current URL, page loading and handling a time event. A macrotask is a well-defined unit of work, and when the event loop has handled a macrotask, it can continue with other tasks. Microtasks are tasks that update the application state and should therefore take priority over macrotasks. These include asynchronous tasks such as *promise callbacks* and changes to the hierarchy of the DOM tree. These are tasks that might require UI rendering after they are completed, and should therefore be handled before a re-rendering task [55, Ch.13].

The exact machinery of the event loop is hard to understand as different browsers implement it differently, and the event loop is not mentioned in the ECMAScript specification [55, Ch.13], but the HTML specification [60] has a detailed description of how a JavaScript event loop should behave. When the event loop checks the event queues, it should check if there are any macrotasks, and handle the first one, and then, while

Listing 2.4: The complexity of nested callback functions.

```javascript
function getData() {
  retrieveDataFromAPI(url, (response) => {
    getDataFromAnotherAPI(response, (newResponse) => {
      showData(newResponse, (text) => {
        console.log(`result: ${text}`);
      });
    });
  });
}
```

the microqueue is not empty, it should handle the microtasks [60, 8.1.7.3]. Thus, during one iteration of the loop, at most one macrotask is handled, and all the microtasks are handled. After the microqueue is empty, the event loop checks if the page needs re-rendering.

2.2.2 Callbacks

In earlier versions of JavaScript, *callbacks* were the main way to perform asynchronous operations without blocking the UI thread. Callbacks are functions passed to other functions as parameters, and their intended use is that the callback will be called once the function's operation is completed. An example of a callback is an event handler, like the function passed to the `onClick` attribute of an HTML button tag. Using callbacks, the GUI can remain responsive while performing some computationally intensive task, because the callbacks are only invoked once the task is finished [55]. Callbacks provide us with a straightforward approach for developing GUIs that not only interact seamlessly with users but also respond promptly to user inputs while handling tasks.

The drawback of relying solely on callbacks to manage asynchronous operations in JavaScript is the readability and flexibility of one's program code. If one only needs one callback function, the resulting code can be understood easily. The problem arises when one wants to execute a series of asynchronous operations sequentially, where each operation depends on the result of the previous. This problem is portrayed in Listing 2.4. To perform several callbacks in a certain order, one calls them "inside of" each other, each using the result of the previous callback to produce the new result. Altering the order of the callbacks is non-trivial, as there are many nested expressions to work with. Further, it might be challenging to understand the code, and it is easy to get lost in the convoluted expressions. This way of chaining sequential asynchronous operations is often referred to as "callback hell" [49, 12].

2.2.3 Promises

Promises avoids this "callback hell" by encapsulating the asynchronous process and giving us a method to decide what should happen after the asynchronous process is completed. In modern JavaScript, promises, which were introduced in 2015 [25], serve as the cornerstone of asynchronous programming. Promises are objects holding the state of an asynchronous operation, and we can instruct the promise on what should follow a change of the state. The state of a promise object is always one of three values; it is either *pending*, *fulfilled*, or *rejected* [43]. When the promise is first created, its state starts as pending. When we *resolve* a promise, we change the state from pending to fulfilled. If we *reject* a promise, the state becomes rejected.

To attach reactions to a promise, that is, react when the promise is fulfilled or rejected, we can use the method `then(onFulfilled, onRejected)`. The parameters `onFulfilled` and `onRejected` are callback functions one provides to define what happens to the value with which the promise is resolved or rejected. The `then` method returns a new promise that we say is *chained* to the promise it was called on. If we resolve the original promise, the `onFulfilled` reaction will then be scheduled for execution immediately.

The `onRejected` parameter is optional; if `onRejected` is present, and the promise is rejected with a value `x`, the promise returned by `then` will be *resolved* with `onRejected(x)`. However, if a chained promise has no reject-reaction, this promise is rejected too. If a promise is rejected and there are no reject reactions downstream the chain, an error will be thrown.

As the `then` method returns a new promise, we can program longer sequential asynchronous operations by calling `then` on the promise returned by the previous `then`, creating a chain of promises. Even though callback functions are still needed to chain promises, the "callback hell" shown in Listing 2.4 can hereby be replaced by the much more elegant and easily managed code in Listing 2.5.

Listing 2.5: Chained promises.

```
retrieveDataFromAPI(url)
  .then((response) => getDataFromAnotherAPI(response))
  .then((newResponse) => showData(newResponse))
  .then((text) => console.log(`result: ${text}`));
```

When we further want to use the result of an asynchronous operation, promises offer another advantage as opposed to callbacks; each promise in the promise chain is a *first-class citizen.* A first-class citizen is an entity that programmers can perform all common programming operations on, such as assigning it to variables, passing it as arguments, and returning it from functions. Asynchronous operations implemented with callbacks are not first-class citizens, because with callbacks we do not obtain a concrete result to refer to. In Listing 2.4 we have to encapsulate the asynchronous operations inside a function to make it possible to refer to later. Even with such a reference, we cannot assign any further reaction to the operation as we can with promises.

2.2.4 Generator functions

Another asynchronous programming technique used in many programming languages is *generators*. A generator is an object returned by a generator function, introduced in ES6 together with promises [29, 25, 55]. We use generators when we want a function to wait to return something until it is prompted to do so. We annotate the function with an asterisk (*) to make it a generator function, as shown in Listing 2.6. Generators introduce the keyword `yield`, which works similarly to `return`, but where `return` returns a value and terminates the function, `yield` returns both a value and the control flow to its caller, thus pausing its own execution.

To retrieve the next value from a generator, we use the method `next` on the generator returned by a generator function. In the generator example shown in Listing 2.6, the `generatorObject` will first return the value {`value: 5, done: false`}. We could then do some operations with this value while the generator is paused. When we need the next value, we call `next` again and get {`value: 4, done: false`}. The next time we call `next`, we get {`value: 3, done: true` }. If we attempt to call `next` again, we will be warned that there are no more values.

Listing 2.6: Generator function.

```
function* generatorFunction() {
  yield 5;
  yield 4;
  return 3;
}
const generatorObject = generatorFunction();
console.log(generatorObject.next());//{value: 5, done: false}
```

The `yield` keyword works by releasing control of the function, yielding the value to the code that invoked the generator. Calling `next` returns the control to the generator, which executes until the next `yield` [35]. A generator function can have many different use cases, a common use case is when we want a function to run indefinitely, like a function generating the next number in a Fibonacci sequence.

Generators are a useful mechanism for handling asynchronous code [55]. If we put the asynchronous code inside a generator, we can then yield whenever we execute an asynchronous task. Outside the generator function, we create a promise that resolves when the `next` method returns a value. This will not cause blocking and unresponsiveness as yielding lets go of the control of the execution context [55].

2.2.5 Programming with `async` and `await`

As a further simplification of the use of promises, the `async` and `await` keywords were introduced in JavaScript. Although long chains of `then`'s are much more elegant than nested callbacks, understanding the chains may still be challenging as we end up presenting several operations over a few lines. The `await` keyword makes it possible to write asynchronous code that looks synchronous and more imperative and recognizable than the promise chains [35].

When we want a program to wait for the value of an asynchronous function that returns a promise, we can label the function as `async`, and then simply write `await` in front of a call to that function. The execution of an `await` statement yields, and is resumed after the promise returned from the `async` function is settled. The promise's value appears as the return value of the `async` function. If the promise is rejected with a value, the value is thrown. In Listing 2.7 we can see how the previous code would look employing `async` and `await` instead of `then`.

Listing 2.7: Chained promises using async/await.

```
async function getData() {
    let response = await retrieveDataFromAPI(URL);
    let newResponse = await getDataFromAnotherAPI(response);
    let text = await showData(newResponse);
    console.log(`result: ${text}`);
}
```

The way `await` works is by releasing control of the thread when it is still awaiting a promise's settlement. The thread then resumes where the `async` function was called from and continues with whatever the next instructions of the program are [29]. When the promise we are waiting for is eventually resolved, the code following the await call gets queued. If the thread is busy with another task, this will not interrupt that task. The way `await` works may seem familiar, as it is very similar to `yield`[2].

2.2.6 Promises and event coalescing strategies

When combining chained promises with event coalescing strategies like debouncing, an issue may arise when promises at the end of a promise chain need resolving, and resolving the first promise is debounced. For the promises at the end of the chain to resolve, they would need to know how to resolve the first promise with the correct value. An example of this can occur in a GUI with a search field, where typing generates search suggestions and pressing the Enter key triggers the search process.

If we want to use promises to manage the event handling of the search field GUI, the flow of the GUI could for example be described as follows. The first promise, `query`, gets resolved with the input from the input field. We then trigger fetching suggestions by chaining `query` with a second promise, `suggestions`, which returns a list of suggestions based on the input. This second promise `suggestions` is then again chained with the operation to show the search suggestions in the GUI. We have a counter to keep track of which suggestion the user picks, with the default choice being the first suggestion. Should the user press Enter after seeing the search suggestions, the first suggestion of the list will then autocomplete the search field with the suggestion and execute the actual search for the input.

Fetching the search suggestion list is the operation that takes the longest here, as it will involve retrieving data from an API. We debounce resolving the first promise `query`, as it is unnecessary to fetch search suggestions for every key press, and resolving `query` will trigger the resolution of the rest of the promises in the chain.

We have an event listener for the Enter key, and when it fires, it chains the second promise `suggestions` with both the autocompletion operation and starting the search

[2]There is some debate about whether `await` is just `yield` with some additional functionality [4]; it is certainly possible to emulate the behavior of `await` using `yield` and promises, as described in Section 2.2.4.

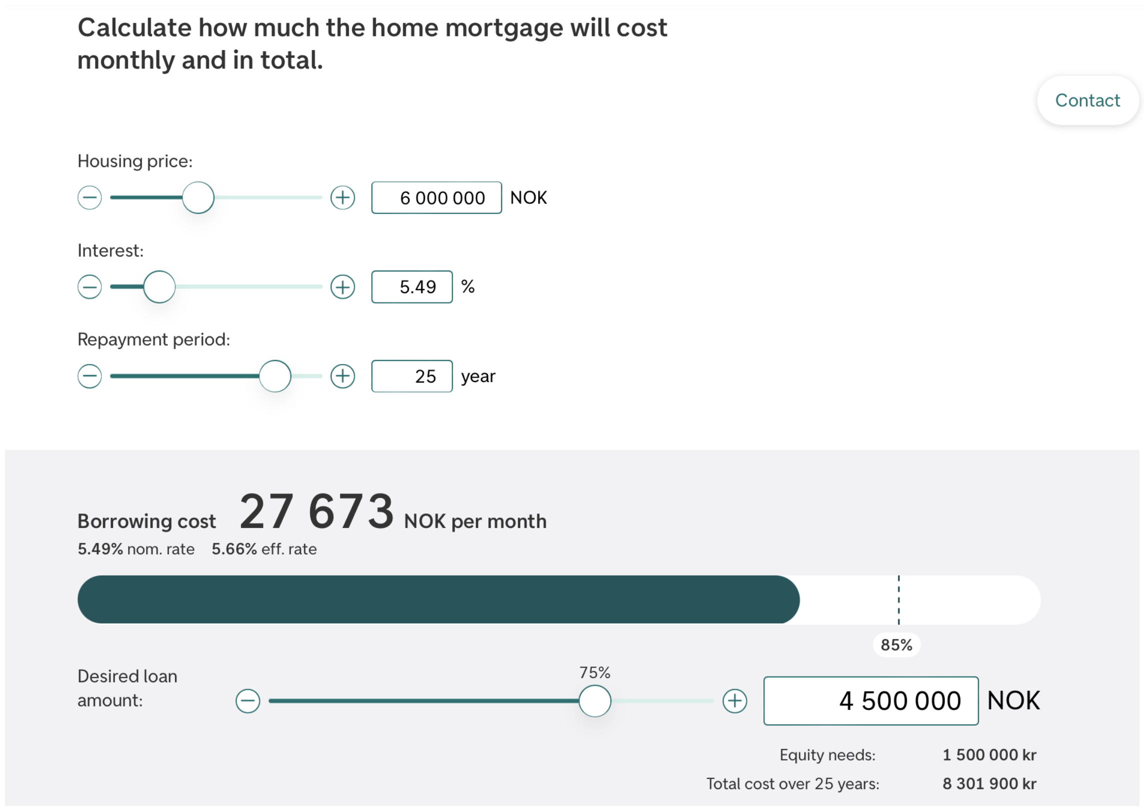
process. If the user types something, and then before the search suggestions appear, they press Enter, we can deduce that they know what the first search suggestion is going to be, and we should start the search process immediately with the new value. What happens in many GUIs instead is that search process commences with the previous value that was input, not the most recent. The problem is that we have employed debouncing, and so the promise chain does not know about the updated value. This results in a search for the wrong input, and the user will have to attempt the search anew. It would therefore be useful with a way to interrupt the ongoing delay, and resolve the first promise `query` with the correct value right away.

2.3 Multi-way dataflow constraint systems

In the previous section, we discussed an example of debouncing causing unwanted behavior in a GUI. This can happen in a lot of different GUIs, and it can be especially detrimental if the GUI has a large network of dependencies from input sources to views that need constant updating. In many different websites, we will encounter a GUI where we input some data, and calculations, graphics or information will change depending on the data that was entered. The different widgets in these websites are related and affect each other, and the relationships require complicated ad-hoc algorithms in each widget's event handler to be maintained [45]. In the GUIs of these websites, we can define the *dataflow execution model* [46] of the program, a directed graph detailing how the different variables affect each other, i.e., their dependencies. Many dependencies going in different directions can lead to a great computational toll on the program and reasoning toll on the programmer, and therefore efforts to reduce these tolls have been attempted in the field of dataflow programming.

An example of one of these web-based GUIs is a house loan calculator. We can find one on the bank DNB's website [6], shown in Figure 2.3. The calculator has three input fields for housing price, interest, and repayment period, and one can also input one's desired loan amount both with the slider and the input field beside the slider.

Many fields in this calculator are uneditable: users cannot alter the monthly payment, equity needs or the total cost. This is surprising, as users may very well know their savings and how much they want to pay on the mortgage a month, and want to know how much they can buy for. Likely one of the reasons why we cannot alter these fields is that this would introduce more than one possible dataflow in the calculator, which would have


Calculate how much the home mortgage will cost monthly and in total.

Housing price: 6 000 000 NOK

Interest: 5.49 %

Repayment period: 25 year

Borrowing cost 27 673 NOK per month

5.49% nom. rate 5.66% eff. rate

Desired loan amount: 75% 4 500 000 NOK

Equity needs: 1 500 000 kr

Total cost over 25 years: 8 301 900 kr

Figure 2.3: DNB house loan calculator.

to be managed in each of the widget's event handler. For example, the total cost is calculated from the monthly cost and the number of months in the repayment period. The calculation of the monthly payment needs the repayment period, the interest rate and the total cost of the loan. If the total cost of the loan is altered, the event handler of the total cost input field would have to accordingly update the monthly cost, the repayment period, or both. It would make the most sense to subsequently alter the monthly cost, but what if the user just altered the monthly cost before editing the total cost? Should we infer that they have decided upon a monthly cost and want to see the other variables change?

Which variables should be updated and which should be left untouched must be dynamically decided by each event handler. It involves introducing extra global variables, e.g., to store the order in which the variables were edited, and error-prone code with branches for every possible scenario. Moreover, if an extra input field is introduced in calculator, the code of all the existing event handlers would subsequently have to be changed.

GUIs with multi-directional dataflow will often therefore exhibit very high levels of code complexity. The development of GUIs with many possible dataflows can be simplified by using multi-way dataflow constraint systems (MDCS) [31]. Here the relations between the variables are specified as constraints, and when variables are updated, the system propagates the change to the other variables, making sure all constraints are enforced. The JavaScript library HotDrink has an implementation of a MDCS that lets the programmer set up variables and constraints for the different relations. Whenever a variable is updated, the constraint system—managed by the library—finds a dataflow and computes new values according to the dataflow. The burden of managing multiple dataflows is shifted from the application programmer to the library.

2.3.1 Event coalescing in HotDrink

When implementing GUIs with complex dataflows, HotDrink simplifies how the programmer can specify the dataflow. To manage the complex dataflows that can arise, HotDrink uses promises graphs: for a dataflow where one variable `x` is dependent on the value of a variable `y` to compute its own value, HotDrink constructs a promise graph where a promise with the value of `x` is chained to a promise with the value of `y`. When the promise representing `y` resolves, it will send its value to the promise representing `x` [31].

Whenever a variable changes, a new promise graph that propagates the edited value to other variables is constructed. Which promise is at the start of the promise chain depends on the dataflow and is therefore not fixed. The library manages the dataflows, and the programmer should not have to know which promises are at the start of the promise chain represented by the promise graphs. However, if the programmer wants to use event coalescing strategies on some widgets' event handlers to avoid calculating the dataflow at every user event, the programmer would have to know which promises are at the start of a promise graph. They would need to identify the first promise of a chain in every promise graph created to avoid promises further down the chain needlessly waiting, as discussed previously in Section 2.2.6. They would subsequently have to make the program "aware" of the event-coalesced promise, and figure out how to resolve the promise immediately if the value is needed.

We need a way to notify the event-coalesced promise in the chain that it no longer has to wait from wherever we need its value. Enabling the interruption of artificially introduced delays—in a safe and modular way—is what this thesis is attempting to solve.

Chapter 3

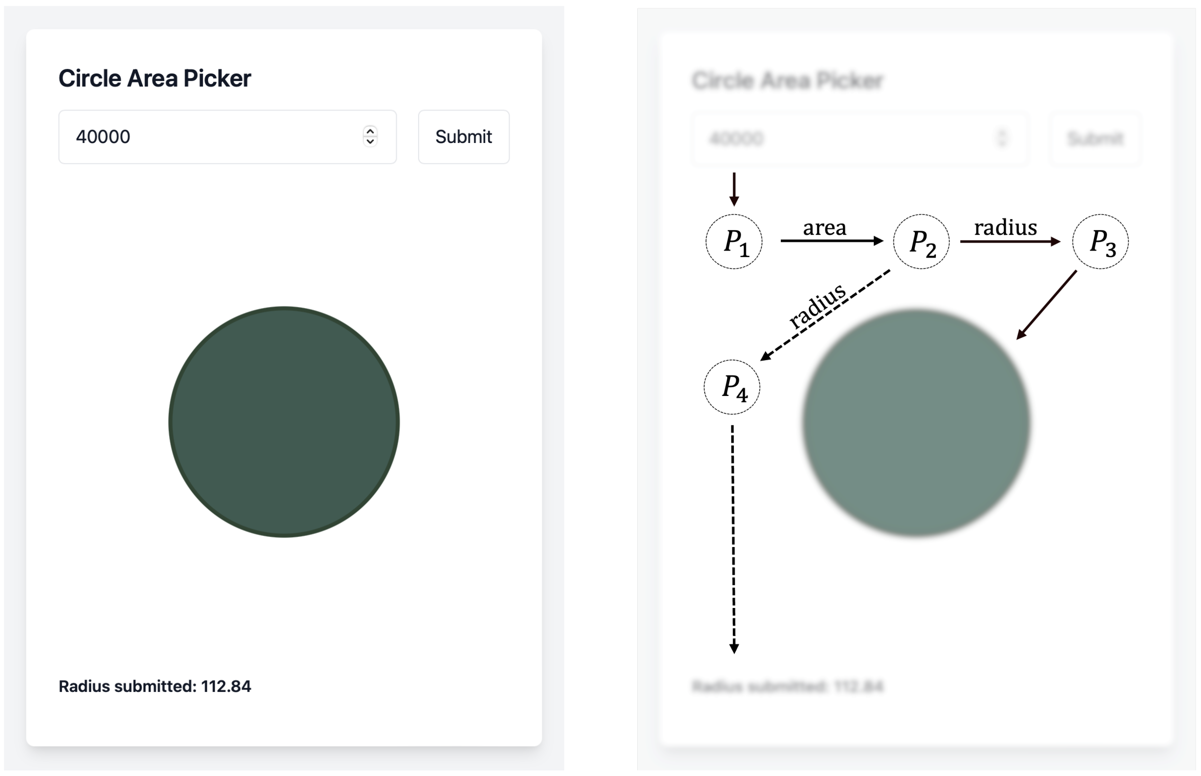
Flushable Promises

In the previous chapter, we discussed methods for developing GUIs that obtain the correct values at the right times while minimizing the computational burden that can stem from unnecessary function calls and processes. In Section 2.2.6, the problem of functions needlessly waiting for values that especially occur when we use promises and debouncing together was briefly described. In this chapter, we describe our solution to the problem, which involves extending the existing functionality of promises. We use a concrete example GUI to frame the discussion.

3.1 A simple circle area picker

A GUI that commonly employs debouncing is a search bar providing search suggestions while the user types. As mentioned in Section 2.1.1, this choice typically stems from the observation that users frequently will know the word they want to search for and type several letters at once. Requesting a server for search suggestions for every letter typed would be redundant.

By moderating the number of function invocations that will result in search suggestions, two key benefits are obtained: Firstly, excessive function calls can reduce the application's performance, leading to slow responsiveness, which is not optimal for the user's experience. Secondly, the continual emergence of new search suggestions while the user is still typing can prove distracting and overwhelming. In our example application—a circle area picker—the second concern is more important. The function triggered by



(a) GUI (b) Promise dependencies

Figure 3.1: Circle area picker GUI and its dependencies.

an event is by itself not computationally expensive, but we still benefit from debouncing to not overwhelm the user. The application is illustrated in Figure 3.1 (a): A simple GUI with a number input field deciding the area of a circle, a "Submit" button, and the resulting circle. If the user clicks "Submit", we simulate sending the data to the server, and a message appears below confirming the radius submitted.

In the "Circle area picker", the user can type a number and the circle will subsequently be resized. If the circle is resized at every keypress, and we want to input the area 130, it would change three times in the time it took to write 130: we would see the circle first change to the area of 1, then to 13, and finally to 130. As resizing the circle is simply re-rendering an HTML element, this is not likely to cause the application to slow down. However, it is not necessary to resize the circle three times, and the constant re-rendering of the circle might very well be distracting. Thus we have to make a choice regarding which keypress events should result in a circle resizing and which should be ignored. It might be enough to wait to change the circle until the user clicks "Submit" or presses Enter. Many applications delay processing input until the input field becomes inactive, i.e., when the user clicks somewhere outside the input field. On the other hand, many GUIs are highly responsive, and the user might expect the circle to change on its own, without having to click somewhere outside the input field. The application could therefore benefit from debouncing the circle resizing function that is called when a keypress event happens.

With debouncing applied to the event handling function, when a user types numbers into the input field, there will be a short delay before the circle gets resized. This will work for most of the use cases of the GUI, but in the case that the user types some numbers and then swiftly presses Enter, the “Submit” button, or somewhere outside the input field, they may expect the circle to be resized immediately and that the most recent user-provided radius value is used in whatever action “submitting” sends the value to. Moreover, the reason for debouncing no longer applies because we can be sure that they have settled on the area of their choice. In this case, we want to interrupt the delay and resize the circle right away.

In Figure 3.1 (b), the GUI’s program flow is illustrated. When a new keypress event from the input field is created, a new promise P_1 is created unless there already exists a pending promise. We chain P_1 with the operation to calculate the radius from the area. The chaining method `then` returns a new promise we refer to as P_2. When P_1 gets resolved with the area value, it then passes the area value to P_2, which calculates and returns the resulting radius. P_2 is then chained with the operation to rerender the page with the new circle, which returns the promise P_3.

In this application, the action of typing a number and waiting to see the circle updating, and the action of typing and clicking “Submit” or Enter, are separate user interactions with different results. When the user clicks “Submit” or Enter, we simulate updating the backend by displaying the radius submitted below the circle. We need the radius to update this text. P_2 returns the radius, so we chain it again with P_4, with the operation to update the text field.

We debounce resolving P_1 with the event value, and so if another event is created while we are in a delay, we postpone resolving P_1. If the user types a value x and then swiftly presses Enter, we want to interrupt the delay caused by the debouncing and get P_2 resolved with x as quickly as possible, as we want to update the text and the circle. The issue is that P_2 will be waiting for P_1 to resolve, and P_1 will be resolved with the correct value after the debouncing delay is finished. If we resolve P_1 right away it would therefore not be with the correct value. We could make another function to retrieve the value and resolve P_1 with it in this case, but the “Submit” button would have to “know” about P_2’s preceding promise, and that the resolution of P_1 is debounced, which makes the code more tangled than it needs to be.

3.2 Flushable promises

To solve the problem of waiting for a value because it itself is waiting for a delay to pass before it can get resolved, we have to look at how the values are connected and how we may ship information between them. In the case of promises in JavaScript, a promise further down in a chain can only be resolved once the preceding promise in the chain gets resolved. This is the rule for all of the promises in the chain. In the case that we have access to an arbitrary promise that we know is not the first promise of a promise chain, and want its value right away, we need to communicate to the promise causing the delay, the first promise in the chain of pending promises, that it should be resolved immediately. We thus need the information to flow backward, or for every promise in the chain to carry information about the previous one. In regular promises, information does not flow backward, and we do not have a pointer to the promise that needs to be resolved for the current promise to be resolved. A possible solution is for every promise that is created with a `then` call, or by some other means that creates a dependency to another promise, to be created with a backward link to the previous promise.

A new type of promise where every promise in a chain has a link to the previous promise can solve part of the problem at hand. It gives us the functionality we need to notify the delay-causing promise that it needs to "hurry up" and resolve so every downstream promise can resolve, in other words, *flush* the promise chain. The act of flushing the promise chain thus entails sending a request upstream that tells whatever promise is at the head of the chain to resolve itself.

Every promise needs to be resolved with a certain value. When rushing a promise chain, we do not specify a value it should resolve to, but expect that the promise knows how to obtain that value. When we construct a "flushable" promise we equip it with a special function, we name it `onFlush`, that will provide this value.

Our solution to the backward link is to simply store the previous promise as an attribute of the current promise. The only promises that will not have this attribute are the first promises of the promise chains. To "travel upstream" in a chain, we call flush recursively on the previous promise. Once we arrive at a promise that does not have a backlink, we know we are at the head of the chain. With the first promise of the chain acquired, we can call the `onFlush` function that we passed to the promise when initializing it, and resolve the promise with the value `onFlush` returns. When we resolve the first promise, every other promise in the chain will be able to resolve and we have achieved our goal.

Listing 3.1: Creating a new flushable promise.

```
function createFlushablePromise() {
  getInputValue = new FlushablePromise(()=>
   ↪ getValueFromNumberPicker());
  getRadius = getInputValue.then(
    (area) => {
      promise_is_settled = true;
      return calculateRadius(area);
      },
    (error) => {
      promise_is_settled = true;
      return error;
    });
  getRadius.then((radius) => updateCircleRadius(radius));
}
```

3.3 Using flushable promises

We call this new type of promise a *flushable promise*. In the example of the circle area picker, with regular promises there was no way to signal P_1 from P_2 to and tell P_1 to resolve with x, but with the flushable promise developed in this thesis, there is. To see how the flushable promise would work in practice, we continue with the example of the circle area picker, examining how it is implemented with the new version of promises.

As the flushable promise is developed to work as similarly as possible to regular promises, the GUI implementation of the circle area picker will look very similar to an implementation that uses regular promises. The difference between the implementations is the use of the flushable promise, and that whenever the program needs a value of a promise at the end of the promise chain, this promise is flushed to obtain said value.

The GUI still has some shared variables for accessing the promises P_1 and P_2, in the code referred to as `getInputValue` and `getRadius`, from different functions. As shown in Listing 3.1, we create a promise chain where we first pass the area from `getInputValue` to `getRadius`, and when the radius has been calculated, we update the circle with another promise. Here we also handle errors if there are any. As mentioned, `getInputValue` and `getRadius` are shared variables declared outside any function and so if they already have a value they will be overwritten here.

The executable we pass as an argument to the constructor of `getInputValue` in Listing 3.1 is the only element in this code snippet that differs from an implementation

Listing 3.2: Function called at every change in input field.

```
function numberChanged(){
  let newValue = getValueFromNumberPicker();
  if (!getInputValue || promise_is_settled) {
    createFlushablePromise();
  }
  debouncedResolvePromise(newValue);
}
```

using regular promises. This is the `onFlush` parameter discussed previously, which here is an arrow function returning the function `getValueFromNumberPicker`. This method retrieves the current value in the number input field.

Whenever the user types something or alters the number in the input field, the function `numberChanged` will be called. Referring to Listing 3.2, we first retrieve the value from the number picker. Note that this is the same function we passed to the `onFlush` attribute on `getRadius`. We then check if `getInputValue` is not yet initialized, or if it already has been settled. Normally we cannot access the state of a promise, and that is not possible with flushable promises either, which is why in Listing 3.1 we set the boolean shared variable `promise_is_settled` to true in both arguments of `then(onFulfilled, onRejected)`. If `getInputValue` is either undefined or settled, we call the function creating a new promise, overwriting the old variable value of `getInputValue` and `getRadius`. If `getInputValue` is neither settled nor undefined, that means that it already has been instantiated and is waiting to be resolved. In either case, we call the function `debouncedResolvePromise`.

Listing 3.3: Function returning debounced function.

```
const debouncedResolvePromise =
  debounce((value) => (resolvePromise(value)), 500);
```

In Listing 3.3, we debounce the `resolvePromise` function and pass it the value from `getValueFromNumberPicker`. In Section 2.1.3, the significance of the correct way to debounce a function was stressed. As mentioned, `debounce` returns a function, which we save in `debouncedResolvePromise`, and it is invoked at every keypress event in the input field. The function that is debounced, `resolvePromise`, resolves `getInputValue` with the value passed to it, and is called from `debouncedResolvePromise` whenever the timeout is allowed to conclude without any more calls to `debouncedResolvePromise`.

Listing 3.4: Function for flushing the promise chain.

```
function submitOrEnterClicked() {
    let submitText = getRadius.then((value)=>updateRadiusText(
  ↪ value))
    submitText.flush();
  }
```

As mentioned in Section 3.1, if the user clicks “Submit” or Enter, we update the text with the radius submitted. To do this, we chain the promise `getRadius` once more, with the operation update the text field. In Listing 3.4, the event handler for the “Submit” button and the Enter keypress action is shown. We chain the second promise returning the suggestions with a new promise `submitText`, responsible for updating the text field. As the user has clicked “Submit” or Enter, we do not want to debounce the resolution of `getInputValue`. If the circle area picker was implemented using regular promises, here we would have to retrieve the value from the input field, and resolve the first promise with that value. Instead we call `flush` on `submitText`. This will cause the entire promise chain to flush, ultimately resolving `getInputValue` with the value returned by `getValueFromNumberPicker`.

Calling `flush` does not directly interrupt the debouncing, it only speeds up the communication between the promises, resolving the promises by immediately resolving the first promise. We do not have to bother to use `clearTimeout` as when `getInputValue` first gets resolved, it cannot get resolved again. This is a property of regular promises, and if we try to resolve `getInputValue` again, which does happen in this application when `submitOrEnterClicked` is used, it is simply ignored.

Chapter 4

Extending λ_p calculus with flushing capabilities

The use of promises is known to be error-prone, possibly because there are no static checks to ensure correct usage; programmers may forget to resolve a promise or call `then` with a non-function argument [12, 49]. If we want an IDE, linter or compiler to be able to tell us the cause of the error in our implementation using promises, we need to perform some type of program analysis. In order to analyze the program in a correct manner, we need to be able to formally reason about promises. The paper "A Model for Reasoning About JavaScript Promises" [49] introduces a formal λ_p calculus for reasoning about promises. The authors' motivation for making this formal language is to develop tools to detect errors in the use of promises. Their "promise calculus" is an extension of an already established λ_{JS} calculus [37], and specifies in detail the processes that occur when the different promise methods are called, both regarding the promise itself, and the event loop.

To be able to reason about the flushable promises, and to specify precisely its semantics, we extend the λ_p calculus with the functionality to flush promise chains. The updated λ_{fp} calculus is developed based on the idea that promises in JavaScript are extended to include a `flush` method. For this we do not need to change the entire specification. In this chapter, we give a brief introduction to the λ_p calculus, and show the rules that change to introduce the flushing ability to λ_p.

This `flush` method should behave as described previously in this chapter; when called on an arbitrary promise that is not the first promise of its chain, it should "travel upstream" the promise chain, until it finds the originators of the chain, and then flush this promise. The originator of the chain can be equipped with a lambda function at the time of its creation; this function will settle the promise. If it was passed a lambda upon creation, and we call flush on an arbitrary promise in its chain, we can resolve the originator with this lambda function, and trigger the resolution of the entire promise chain.

4.1 Expressions and state in λ_{fp}

The λ_p calculus we extend consists of expressions and evaluation rules. The runtime state is a tuple, whose components represent different aspects of an application's state, such as addresses allocated in the heap or whether a promise is pending, fulfilled or rejected. As the λ_p calculus has a "small-step reduction semantics" [49], the evaluation rules specify how "holes" in expressions are reduced by updating the runtime state. Holes in expressions in evaluation contexts represents placeholders for subexpressions. When we are evaluating an expression, the hole in the expression is where we want to further reduce or substitute the expression [30]. A hole is represented by a $\square$ here. An evaluation rule states how an expression can be reduced, and how the reduction affects the runtime state. We now present the updated syntax, evaluation context and runtime state, where we have highlighted our additions to the established λ_p

The updated syntax is shown in Figure 4.1 (a) and consists of seven expressions: `promisify` evaluates an object and turns it into a pending promise; `resolve` fulfills a pending promise with a value and schedules the promise's resolve reactions; `reject` rejects a pending promise with a value, and schedules the promise's reject reactions; the `onResolve` expression registers a new resolve reaction and creates a new promise dependent on the original promise, and `onReject` registers a new reject reaction and creates a new promise also dependent on the original promise; `link` links two existing promises, such that if the first promise resolves, the second will also get resolved, with the same value as the first promise—this applies also if the first promise gets rejected [49]. The only additional expression that is new in λ_p is `flush`. Calling `flush` on a pending promise entails flushing the promise chain of that promise, i.e., following the backlink to the first promise of a promise chain, and invoking the function passed during the promise's creation.

The extended evaluation context of λ_{fp} is shown in Figure 4.1 (b). As previously stated, the evaluation context explains how sub-expressions can be reduced. We show two ways of calling an expression, stating that we can both call for example `onResolve`

$$
\begin{array}{lll}
e \in \mathit{Exp} & = \texttt{promisify}(e, \texttt{e}) & \text{[create promise]} \\
& \mid e\texttt{.resolve(e)} & \text{[resolve promise]} \\
& \mid e\texttt{.reject(e)} & \text{[reject promise]} \\
& \mid e\texttt{.onResolve(e)} & \text{[chain promise]} \\
& \mid e\texttt{.onReject(e)} & \text{[chain promise]} \\
& \mid e\texttt{.link(e)} & \text{[link promises]} \\
& \mid e\texttt{.flush()} & \text{[flush promise]}
\end{array}
$$

(a)

$$
\begin{array}{ll}
E & = \square \\
& \mid \texttt{promisify}(E, e) \mid \texttt{promisify}(e, E) \\
& \mid E\texttt{.resolve}(e) \mid v\texttt{.resolve}(E) \\
& \mid E\texttt{.reject}(e) \mid v\texttt{.reject}(E) \\
& \mid E\texttt{.onResolve}(e) \mid v\texttt{.onResolve}(E) \\
& \mid E\texttt{.onReject}(e) \mid v\texttt{.onReject}(E) \\
& \mid E\texttt{.link}(e) \mid v\texttt{.link}(E) \\
& \mid E\texttt{.flush}()
\end{array}
$$

(b)

Figure 4.1: (a) Syntax of λ_{fp}. (b) Evaluation contexts for λ_{fp}.

$$
\begin{array}{rl}
\sigma \in \mathit{Heap} & = \mathit{Addr} \hookrightarrow \mathit{Val} \\
\psi \in \mathit{PromiseState} & = \mathit{Addr} \hookrightarrow \mathit{PromiseValue} \\
f \in \mathit{FulfillReactions} & = \mathit{Addr} \hookrightarrow (\mathit{Reaction} \times \mathit{Addr})^* \\
r \in \mathit{RejectReactions} & = \mathit{Addr} \hookrightarrow (\mathit{Reaction} \times \mathit{Addr})^* \\
\pi \in \mathit{Queue} & = (\mathit{PromiseValue} \times \mathit{Reaction} \times \mathit{Addr})^* \\
\rho \in \mathit{Reaction} & = \mathit{Lam} \mid \mathit{default} \\
\alpha \in \mathit{FlushReaction} & = \mathit{Lam}_0 \mid \texttt{null} \\
\Psi \in \mathit{PromiseValue} & = \{\mathsf{P}, \mathsf{F}(\mathit{Val}), \mathsf{R}(\mathit{Val})\} \\
\varphi \in \mathit{RetrieveReactions} & = \mathit{Addr} \hookrightarrow \mathit{FlushReaction} \\
\delta \in \mathit{PrecedingPromises} & = \mathit{Addr} \hookrightarrow \mathit{Addr}^*
\end{array}
$$

Figure 4.2: Runtime state for λ_{fp}.

on an expression that could be further reduced, or we can pass a reducible expression to `onResolve`.

The updated runtime state of our λ_{fp} calculus is shown in Figure 4.2. In the runtime state of λ_p, an *address* refers to a location or a memory address. The *heap* σ refers to the heap of the program mapping addresses to values. The *promise state* ψ maps addresses to *promise values* Ψ, which can be either pending P, fulfilled $\mathsf{F}(v)$, or rejected $\mathsf{R}(v)$, where v is the value settling the promise. The *fulfill-* and *reject reactions* f and r map addresses to a list of pairs of a *reaction* ρ and the dependent promise address. The reaction can either be a lambda or a default function, which can be thought of as a function that just returns its argument. Finally, the *queue* π denotes the event queue, where we put scheduled reactions that the event loop will execute at some point.

We introduce three additional components to the context of λ_{fp} that are not in λ_p: The *flush reactions* α, denoting valid types of functions passed to `promisify`; *retrieve reactions* maps the address of a promise to a flush reaction; and lastly, the *preceding promises* maps the address of one promise to the address of one or several promises. The flush reactions can either be a nullary function in Lam_0 or null. The program state can be represented as $\langle \sigma, \psi, f, r, \pi, \varphi, \delta \rangle$. As we introduce new components in the program state, every operation rule in λ_p needs to be updated, but here we only show the rules that are significantly different, that is, more than just updating the program state. The rest of the rules can be found in Appendix A.

4.2 Rules in λ_{fp}

Following are the evaluation rules we either created or made significant changes to in order to extend the λ_p calculus with the flushing ability. In the existing rules that we needed to make adjustments to, we have highlighted these adjustments.

E-Promisify

$$\frac{\begin{array}{ccccc} a \in Addr & a \in \mathsf{dom}(\sigma) & a \notin \mathsf{dom}(\psi) & a \notin \mathsf{dom}(\varphi) & \psi' = \psi[\ a \mapsto \mathsf{P}\ ] \\ f' = f[\ a \mapsto Nil\ ] & r' = r[\ a \mapsto Nil\ ] & \boxed{\delta' = \delta[\ a \mapsto Nil\ ]} & \boxed{\varphi' = \varphi[\ a \mapsto \lambda_0\ ]} & \end{array}}{\langle \sigma, \psi, f, r, \pi, \boxed{\varphi, \delta}, E[\ \mathtt{promisify}(a, \boxed{\lambda_0})\ ] \rangle \rightarrow \langle \sigma, \psi', f', r', \pi, \boxed{\varphi', \delta'}, E[\ a\ ] \rangle}$$

[E-Promisify]. This rule applies when a new promise is created. To be able to use the flush functionality later, we store the promise created and its eventual flush function together. Specifically, we assert that a is an address in the heap, λ_0 is a flush reaction, and a is not previously in the promise state or the flush state. If the expression to be reduced is `promisify`, we initialize the promise state of a to pending P and we initialize the fulfill and reject reactions to the empty list as the promise has no registered reactions upon creation. We also initialize the preceding promises to the empty list, as every promise starts out with their preceding promises as an empty list. We update the retrieve reactions with a mapping from a to λ_0. This is to eventually retrieve the function λ_0 and invoke it if the promise chain is flushed at a later time.

When we create a promise, if we want it to be able to flush and resolve with a value at a later point, we pass a nullary lambda function that retrieves the value as the λ_0

parameter in $\texttt{promisify}(s, \lambda_0)$. If we do not need the flushing functionality, we pass `null`.

The [E-Promisify] rule in λ_{fp} differs from λ_p by having an additional argument passed to `promisify`, verifying that a is not already in the retrieve reactions, and adding a to the retrieve reactions with λ_0.

E-OnResolve-Pending

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \\ \psi(a) = \mathsf{P} \qquad a' \in \mathit{Addr} \qquad a' \notin \mathsf{dom}(\sigma) \qquad \psi' = \psi[\ a' \mapsto \mathsf{P}\ ] \qquad \sigma' = \sigma[\ a' \mapsto \{\}\ ] \\ f' = f[\ a \mapsto f(a) ::: (\lambda, a')\ ][\ a' \mapsto \mathit{Nil}\ ] \qquad r' = r[\ a' \mapsto \mathit{Nil}\ ] \qquad \delta' = \delta[\ a' \mapsto [\ a\ ]\ ] \end{array}}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a.\texttt{onResolve}(\lambda)\ ] \rangle \rightarrow \langle \sigma', \psi', f', r', \pi, \varphi, \delta', E[\ a'\ ] \rangle}$$

[E-OnResolve-Pending]. This rule handles registering a resolve reaction to an existing pending promise. This reaction is what will happen when the promise gets resolved. We can register countless resolve- and reject-reactions for a promise, and all of them will be scheduled to run when said promise is resolved. Specifically, if the expression to be reduced is $a.\texttt{onResolve}(\lambda)$, where a is an address already allocated in the heap, that is, an existing pending promise, and λ is a fulfill reaction, a new promise a' is created. The new promise a' is initialized to the pending state. The fulfill- and reject reactions are initialized to empty lists. To the fulfill reactions of the existing promise a, the pair (λ, a') is added, such that it can be run once a gets resolved. A mapping from a' to a singleton list containing a in the preceding promises is added as well. This is where we create the backlink to the preceding promise. Note that there is no difference between promises that have a flush reaction and promises that do not have a flush reaction, we simply add every chained promise to the preceding promises.

This rule differs only from the original [E-OnResolve-Pending] in the λ_p calculus in that we have added a mapping from a' to a in the preceding promises. [E-OnReject-Pending] would conceptually be very similar to this rule, and we have therefore decided not to present it.

E-Flush-Pending-HasFetchValue

$$\frac{a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \varphi(a) = \lambda_0 \qquad \psi(a) = \mathsf{P}}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a.\texttt{flush}()\ ] \rangle \rightarrow \langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ \texttt{if}(\lambda_0 \neq \texttt{null})\ a.\texttt{resolve}(\lambda_0())\ ] \rangle}$$

[E-Flush-Pending-HasFetchValue]. This rule does not have an equivalent rule in the λ_p calculus. It handles the case where we call flush on a pending promise, and that promise is in φ, which signifies that it is at the head of the promise chain and has a function that can immediately find a value for the promise. If the expression to be reduced is a.`flush()`, and a is an address allocated in the heap, and its promise state is pending, and a is in φ, we have a promise that is at the head of its chain. As λ_0 either can signify a nullary lambda function or null, the returned expression needs to verify this. If it is not null, that means that a was passed a nullary lambda function upon creation. We then resolve a with λ_0. Notice that nothing in the program state has been altered yet. See the rule [E-Resolve-Pending] in Appendix A for what will follow in the case that λ_0 is not `null` and we call a.`resolve`($\lambda_0()$). However, if λ_0 is equal to `null`, nothing will happen in the case that `flush` is called.

E-Flush-Pending-HasPrecedingPromises

$$\frac{\begin{array}{c} a_1 \in \mathit{Addr} \ldots a_n \in \mathit{Addr} \qquad a_1 \in \mathsf{dom}(\sigma) \ldots a_n \in \mathsf{dom}(\sigma) \qquad \psi(a_1) \ldots \psi(a_n) = \mathsf{P} \\ a_k \in \mathit{Addr} \qquad a_k \in \mathsf{dom}(\sigma) \qquad \delta(a_k) = a_1 \ldots a_n \qquad a_k \notin \mathsf{dom}(\varphi) \qquad \psi(a_k) = \mathsf{P} \end{array}}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a_k.\texttt{flush()}\ ] \rangle \rightarrow \langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a_1.\texttt{flush();} \ldots a_n.\texttt{flush();}\ ] \rangle}$$

[E-Flush-Pending-HasPrecedingPromises.] This rule also has no equivalent in the λ_p calculus. It handles the case where `flush` is called on a pending promise a_k that has one or several preceding promises. We refer to the promise or promises that another promise is dependent on as its preceding promises here. A promise may have several preceding promises if it was created with the method `Promise.all`, which we will cover later in this chapter.

Specifically, the expression to be reduced is a_k.`flush()` and a_k is a pending promise address allocated in the heap. $a_1 \ldots a_n$ are also all pending promises that are allocated in the heap. If there is a registered mapping from a_k to $a_1 \ldots a_n$, we call `flush()` for every promise that a_k is mapped to in δ. This rule will then for every preceding promise either call itself, or [E-Flush-Pending-HasFetchValue] if it is the case that we have arrived at the head of the promise chain for one or several of $a_1 \ldots a_n$.

E-Link-Pending

$$\frac{\begin{array}{c} a_1 \in \mathit{Addr} \qquad a_1 \in \mathsf{dom}(\sigma) \\ a_2 \in \mathsf{Addr} \qquad a_2 \in \mathsf{dom}(\sigma) \qquad \psi(a_1) = \mathsf{P} \qquad f' = f[\ a_1 \mapsto f(a_1) ::: (\text{default}, a_2)\ ] \\ r' = f[\ a_1 \mapsto r(a_1) ::: (\text{default}, a_2)\ ] \qquad \boxed{\delta' = \delta[\ a_2 \mapsto [\ a_1\ ]\ ]} \end{array}}{\langle \sigma, \psi, f, r, \pi, \boxed{\varphi, \delta,} E[\ a_1.\texttt{link}(a_2)\ ] \rangle \rightarrow \langle \sigma, \psi, f', r', \pi, \boxed{\varphi, \delta',} E[\ \mathsf{undef}\ ] \rangle}$$

[E-Link-Pending]. This rule handles the specific case when the to-be registered resolve reaction in itself returns another promise. Whatever happens to one promise should then also happen to the promise "linked" to it. Specifically, if a_1 and a_2 are promise addresses allocated in the heap, and a_1 is pending, we update the fulfill- and reject-reactions of a_1 so that if a_1 gets resolved or rejected, a_2 will also get resolved or rejected. We do this by passing the reaction $(\textit{default}, a_2)$ to both reaction lists. We also update the preceding promises to be able to flush the resulting promise chain, as if whatever happens to a_1 also should happen to a_2, we deduce that if a_2 is flushed, a_1 should also be flushed.

This rule only differs from the original [E-Link-Pending] in the λ_p calculus in that we add a mapping from a_2 to a_1 in the preceding promises.

4.3 The semantics of `Promise.all` in λ_{fp}

The Promise API static method `all` takes a sequence of promises and either returns a promise fulfilled with an array of the results of the now fulfilled sequence of promises, or it returns a promise rejected with the value of the first promise to be rejected. It is another way of chaining promises besides `then`. In the λ_p paper, `all` does not have its own rule, its implementation is instead described in words. The method `then`'s implementation is presented, using the other primitive promise operations in λ_p, showing that it is possible to implement with the existing rules. This applies to `then` with our rules as well, but implementing `all` with the operations in our λ_{fp} calculus requires some extra consideration.

To implement `all`, we make a counter and a new promise. We call `then` on every promise passed in the sequence, and inside `onFulfilled`, we add the resolve-value to a list, decrement the counter and check if the counter is zero. If so, we resolve the new promise with the list of resolve-values.

We have to consider how the promise dependencies would look like, and how we mitigate flushing if we have a promise P_i dependent on the result of `all`, which we will refer to as P_k. One or several of the promises $P_1 \dots P_n$ passed as arguments to `all` may be resolved quicker if we flush, and if we call flush on P_i, we want every flushable promise to also flush. We therefore need to set every promise in $P_1 \dots P_n$ as preceding promises for P_k.

Listing 4.1: `all` implemented with λ_{fp} operations.

```
1   Promise.all =
2     function([promise_1 ... promise_n]) {
3       let counter = n;
4       let arr = [];
5       let promise = promisify({});
6       promise.setPrecedingPromises([promise_1 ... promise_n])
7       for (let i = 0; i < n; i++) {
8         promise_i.onResolve((v) => {
9           arr[i] = v;
10          if(--counter == 0) {
11            promise.resolve(arr);
12          }
13        },(e)=>promise.reject(e))
14      }
15      return promise
16    }
```

We have no separate rule for setting preceding promises. One possibility is to make several rules for the process of the `all` method, one for when it is called, to show that we would call `onResolve` on every promise, and one to show what follows when we know we are resolving the last promise and all other promises are already resolved. The issue with separating the rules is that there is no conceivable proper way to imply that the promises in question had previously been passed to `all`. We would have to make another set in the state, just to state that this promise has previously been called in a sequence to `all`, and we would like this to happen when it resolves.

In this thesis we have decided to show how to implement `all` with the existing rules, assuming there exists a method `setPrecedingPromises` that sets a promise's list of preceding promises—such a method is straightforward to implement. Calling `then` registers a resolve reaction that adds a new preceding promise connection, but in the case of `all`, it would not connect the promises we need to be connected; `then` called on P_1 returns a new promise P_2, and P_1 is P_2's preceding promise, while here we want $P_1 \dots P_n$ as preceding promises for P_k, where P_k is a new promise we create in `all`. Nevertheless, Listing 4.1 shows an implementation of `all` with our existing rules, except the use of `setPrecedingPromises` line 6, which is not a standalone operation in our λ_{fp} calculus.

Chapter 5

Implementation

In the previous section, the idea of the flushable promise was described. Furthermore, an example was presented to outline the expected behavior of flushable promises. In this chapter, we describe in detail how flushable promises are implemented.[1]

The difference between a flushable promise and a regular promise is small. The only added functionality is the `flush` method, but to make flushing work we need to have supplemental attributes and private methods as well. As stated in Section 3.2, to accomplish the flushing ability, we need a backward link to the previous promises and a way to let the first promise of a chain know how to retrieve its value immediately. We get these by storing the previous promise as a new attribute, and by providing the option to pass an `onFlush` attribute to a new promise when initializing it.

To implement flushable promises, we made a “promise wrapper class” in JavaScript. There are other ways to add functionality to an existing class in JavaScript, but this method seemed the most straightforward. Hence, the `FlushablePromise` class is a JavaScript class with a regular promise as one of the attributes. Another implementation approach could be to implement promises from the ground up, encoding every rule formalized in λ_{fp}. A such implementation would, however, not take advantage of all the existing functionality of JavaScript promises, like updating the state and chaining promises. Furthermore, any JavaScript engine would likely treat promises from a such implementation differently than regular promises.

Instead of plain JavaScript, we give the definitions in TypeScript. We have implemented flushable promises in both the major gradually typed JavaScript extensions, TypeScript and Flow.

[1]The implementation in its entirety can be found at `https://github.com/MariaBonde/FlushablePromise`.

Listing 5.1: The fields of a flushable promise.

```
class FlushablePromise<T> {
  private promise: Promise<T>;
  public resolve: (x: T) => void;
  public reject: (x: any) => void;
  private fetchValue: () => T;
  private precedingPromises: FlushablePromise<any>[];
  ...
}
```

In the class declaration listed in Listing 5.1, a class called `FlushablePromise` is defined. It is a generic class, as promises should work with values of any type, here identified as `T`. The different attributes have their respective uses, and most of them are self-explanatory. If the value of an instance of the `FlushablePromise` class has the type `T`, then the value of `promise` has to also have the type `T`, and the same applies to the argument for `resolve`. The attribute `fetchValue` is where we store the `onFlush` function if it is given. The value returned from `fetchValue` has to also be the type `T`, as it is what we are going to resolve the promise with. When we reject a promise, the value we reject it with can be anything, and therefore the type has to be `any`.

The `precedingPromises` attribute is a list of flushable promises that can have `any` type. It is an array because a promise can have multiple preceding promises. In general, one should avoid using the type `any` when possible as if we have one `any` type, we often end up having to accept the `any` type everywhere. The values of the promises in `precedingPromises` is not, however, used anywhere in this class, we just use the promises to travel "upstream" the promise chain. Therefore using `any` here does not lead to loss of type information here.

The attributes `promise`, `fetchValue` and `precedingPromises` are private, while `resolve` and `reject` are public. This is because `fetchValue` and `precedingPromises` are only for making flushing possible, and not for external use. The `promise` attribute is private as it should not be necessary to use it externally.

5.1 The `FlushablePromise` constructor

In Section 3.3, we saw the constructor of the flushable promise in use. We passed an arrow function returning another function in the constructor. This parameter is called

Listing 5.2: The constructor in `FlushablePromise`.

```
constructor(onFlush?: () => T) {
  this.promise = new Promise((resolve, reject) => {
    this.resolve = (v) => {
      resolve(v);
    };
    this.reject = (e) => {
      reject(e);
    };
  });
  if (onFlush) {
    this.fetchValue = onFlush;
  }
  this.precedingPromises = []
}
```

`onFlush` and we can see how it is set among the other attributes of the `FlushablePromise` class in Listing 5.2. The `onFlush` parameter is an optional parameter, and we therefore check if it was passed before we set `this.fetchValue` with it. If we do not pass an `onFlush` argument, the promise should behave like a regular promise.

When programmers are introduced to promises through examples, they encounter the promise constructor a lot. When programming in practice, creating a promise using its constructor is, however, not that common. Promises are rather often returned by functions we commonly use. For example, a common way to retrieve data from an API is by using the global `fetch` method. One often uses the response from the API by, e.g., `fetch(url).then(response=>handleData(response))`. Programmers may write such code without really thinking about promises, even though a promise is created and returned by `fetch`, and fulfilled when the response from the API becomes available [61].

Two notable attributes also set in the constructor are `this.resolve` and `this.reject`. When creating a regular promise, as we do on the first line inside the constructor in Listing 5.2, we have access to `resolve` and `reject`, and we can state conditions for under which the promise should get resolved and rejected. Here we exploit this by setting the class attributes `this.resolve` and `this.reject` to these functions. This allows us to resolve and reject the flushable promise instances inside other class methods as well. This method of "extracting" the `resolve` and `reject` methods us quite common, and was already employed in another promise wrapper class in HotDrink, which served as inspiration for the basic structures of the `FlushablePromise` class.

5.2 `then(onFulfilled, onRejected)`

In regular promises, the `then` method registers a reaction to a promise and returns a new, chained promise that again can be chained in order to further register another reaction. In the `FlushablePromise` class, the semantics of the method `then` only differs in that when we register a reaction to a flushable promise, which produces a new promise, flushing the new promise should flush the original promise. This is not straightforward, as the method is overloaded and can be used in many different ways.

The implementation of `then` for flushable promises is given in Listing 5.3. In this section, we explain the machinery of the `then` method for promises and how our implementation makes promises aware of what their preceding promises are.

5.2.1 Type parameters

The method declaration of `then` introduces the type parameters `ResolveType=T` and `RejectType=never`. The type `ResolveType` will be the type of the value of the new promise, should it get resolved. We declare that the default type for the value of the new promise is `T`.

We also introduce `RejectType` and set it to be the default type `never`, which is a type that represents the absence of something, similar to `null` and `undefined`. When a function returns `never`, it usually signifies that they never will return, like a function designed to throw an exception [20]. Here we use `never` as the default type for `RejectType` because if we have not stated a reject type, the type of the value of the returned promise will be `ResolveType | never`, which is computed by the program to just `ResolveType`.

The `onFulfilled` parameter has the type `(a:T) => ResolveType`. The parameter `a` is the value of the previous promise and has type `T`. `ResolveType` is the new type, which is decided by what the programmer introduces here. `onRejected` has the type `(a:any) => RejectType`, as the promise can be rejected with any value. As mentioned in Section 2.2.3, `onRejected` is an optional parameter, and this is because it is not compulsory to provide both a resolve- and reject-reaction. If there is no `onRejected` argument, and the promise gets rejected, it will pass the state of the promise to the next promise in the chain, until it either finds a “handler” for the state [29], or an error is thrown. The `then` method returns a flushable promise with either the `ResolveType` or `RejectType`.

Listing 5.3: Method `then` in FlushablePromise.

```
then<ResolveType = T, RejectType = never>(
  onFulfilled: (a: T) => ResolveType,
  onRejected?: (a: any) => RejectType
): FlushablePromise<ResolveType | RejectType> {
  var newPromise = new FlushablePromise<ResolveType |
↪ RejectType>();

  newPromise.setPrecedingPromise(this);

  this.promise.then(
    (x) => {
      let fulfilledVal = onFulfilled(x);
      if (fulfilledVal instanceof FlushablePromise) {
        fulfilledVal.setPrecedingPromise(this);
        newPromise.setPrecedingPromise(fulfilledVal);
        fulfilledVal.resolve(x);
        return newPromise.resolve(fulfilledVal);
      } else {
        return newPromise.resolve(fulfilledVal);
      }
    },
    (y) => {
      if (onRejected) {
        let rejectedVal = onRejected(y);
        if (rejectedVal instanceof FlushablePromise) {
          rejectedVal.setPrecedingPromise(this);
          newPromise.setPrecedingPromise(rejectedVal);
          rejectedVal.reject(y);
          return newPromise.resolve(rejectedVal);
        } else {
          return newPromise.resolve(rejectedVal);
        }
      } else {
        return newPromise.reject(y);
      }
    }
  );
  return newPromise;
}
```

5.2.2 Achieving the flushing functionality

Replicating the then function of the Promise class in this promise wrapper class requires some consideration, as it should work exactly like a regular promise, but the new promise needs to inherit the attribute precedingPromises. To replicate the functionality of then, we could just have returned this.promise.then() which returns a promise. However, that would return a regular, dependent promise, and we need to alter the dependent promise's attributes. We need every promise in the chain to be a flushable promise. Therefore, we first create a new flushable promise inside then. We then use the private method setPrecedingPromise to set the current promise as a preceding promise for the new promise.

We could check if the current promise either has a current fetchValue or precedingPromises before setting the preceding promise, as we only need the promises that were created with an onFlush, or have a preceding promise that was created with an onFlush, to be flushable. This would, however, give us problems later with the other chaining methods. Because of this, we simply let every chained promise have a preceding promise, and we refer to every object that is an instance of the FlushablePromise class as a flushable promise. If calling flush on a promise has an effect, however, is dependent on whether it or its preceding promise was initialized with an onFlush argument.

5.2.3 Chaining the promise

We use the functionality of then by calling it on the current promise (this). What is inside the following block will only run once this is resolved or rejected. In the arguments of then, we get to decide what happens once this is resolved or rejected, namely onFulfilled and onRejected, respectively. In the first argument, we set the new promise to resolve with the value returned by onFulfilled applied to x. If the current promise gets resolved, we will resolve the next promise in the chain with the onFulfilled provided by it.

In the second argument, we decide what happens if the promise is rejected; we first check if onRejected is defined, and if it is, we resolve the new promise with the result of onRejected applied to y. This might seem counterintuitive, but if we add a way to handle a rejection, we do not want an error to be thrown and the program to conclude, instead we want the chain to continue with the value provided by the onRejected function. If there is no `onRejected`, we reject the new promise with the value sent in. In the case that a promise with no reject reaction in a promise chain gets rejected, every promise in the promise chain will get rejected with the same value until it finds a reject reaction. If there is a promise with a reject reaction in the chain, the promise will get resolved with the result of the reject reaction applied to the value from the first rejection [29].

5.2.4 Linking two promises

There is a special case we need to facilitate when replicating `then`; if `onFulfilled` or `onRejected` returns a promise: Firstly, there is the factor of what follows when this happens in regular promises. "ECMAScript 2025 Language Specification" [27] does not have clear instructions regarding the different return values of `onFulfilled` and `onRejected`, but the book "JavaScript Async: Events, Callbacks, Promises and Async Await" [29] covers a few different cases, depending on what these "handlers" return. If the handler returns a pending promise, it states that "resolution/rejection of the Promise returned by then will be the same as the resolution/rejection of the promise returned by the handler" [29], i.e. that we should do the same to the promise returned by `onFulfilled`/`onRejected` as we do the promise we return in `then`.

To rectify this, inside the `then` method of the `FlushablePromise` class we can take advantage of the fact that we can intercept what happens before the handlers are called inside the "real" `then`. In the `onResolve`-block, before we resolve the current promise, we call `onFulfilled` on the argument `x` sent in by the previously resolved promise, and then we check if the resulting variable is an instance of the `FlushablePromise` class. If it is not a flushable promise, we can just call resolve on the next promise in the chain with the result from `onFulfilled`. On the other hand, if it is the case that the programmer has sent in another flushable promise as the return value for `onFulfilled` or `onRejected`, we need to consider how the flushing ability should work in the resulting chain.

In "A Model for Reasoning About JavaScript Promises" [49], they refer to this mechanism as "linking" two promises, and in the rule, it is defined as simply adding the promise linked to the other promise's fulfill-reactions. We must consider the programmer's possible motivation for linking two flushable promises P_1 and P_2: if the two promises are linked, and we want whatever happens to P_1 to also happen to P_2, and we instantiated P_1 with the `onFlush` argument, then it should also be possible to flush P_2 or a promise chained to P_2 and subsequently resolve P_1. Therefore P_2 should have P_1 as a preceding promise.

There is one more promise to consider here, namely the promise we return in `then`, in Listing 5.3 referred to as `newPromise`; we have previously put `this`, which in this case refers to P_1, as the preceding promise for `newPromise`, but now `this` is the preceding promise of P_2. The `newPromise`, or P_3 also needs a preceding promise if we want to be able to flush the chain, and therefore we set the preceding promise of P_3 to be P_2. We could set P_1 as the preceding promise of both P_2 and P_3, but then that would indicate that they are two separate chains. If P_1 gets resolved, that should lead to both P_2 and subsequently P_3 getting resolved. We believe that if the programmer links P_1 and P_2, chains the resulting P_3 further, and later tries to call P_n`.flush()`, where P_n is in the same chain as P_3, they expect P_1 to be able to call `flush`.

When we are inside the `onResolve`-block, we are handling the case that the previous promise P_1 is already resolved. In that case, we want to also resolve the `newPromise` or P_3, but now there is a promise between them in the chain, namely P_2. Therefore, we have to make sure to not only resolve P_3, but also P_2. The rule is that if P_1 and P_2 are linked, then resolution/rejection will be the same for both of them, therefore we can resolve P_2 with the same `x` that P_1 got resolved with. In the case that P_1 gets rejected, and P_2 is passed as what `onRejected` returns, then we infer that P_2 also should get rejected, but P_3, or `newPromise`, should get resolved with the `onRejected` value. Note that like regular promises, if we resolve a promise with another resolved promise, the promise will get resolved with the value of the already resolved promise [29].

We now have called `then` on our current promise, created a new promise, instructed the resolve- and reject reactions for our new promise, and finally, we return the new promise for further chaining or other use cases. The whole `then` is listed in Listing 5.3.

5.3 `catch(onRejected)` and `finally(onFinally)`

A promise chain needs some kind of error handling, because if there is none and a promise gets rejected, an error will be thrown because of the unhandled promise rejection. If a promise early in the chain gets rejected, the subsequent `onFulfilled` reactions on every promise in the chain are not executed until a `onRejected` is found. In the meantime the value from the rejected promise is simply passed on. If there is an `onRejected`, the rejected value will be its input, and the value from the `onRejected` operation will be passed on in the promise chain. In the case that we do not want a value from an `onRejected` passed on, it is common to only provide resolve reactions and then at the

Listing 5.4: Method `catch` in FlushablePromise.

```
catch<RejectType>(
  onRejected: (x: any) => RejectType
): FlushablePromise<T | RejectType> {
  return this.then((x) => x, onRejected);
}
```

Listing 5.5: Method `finally` in FlushablePromise.

```
finally(onFinally: () => void | any): FlushablePromise<any> {
  if (typeof onFinally != "function") {
    return this.then(onFinally, onFinally);
  } else {
    return this.then(
      (value) => FlushablePromise.resolve(onFinally()).then
 ↪ (() => value),

      (reason) =>
        FlushablePromise.resolve(onFinally()).then(() => {
          throw reason;
        })
    );
  }
}
```

end of the chain write a *catch all error routine* [29]. It is therefore customary to write `catch(onRejected)` at the end of the promise chain, to make sure we handle all errors that might arise. `catch(onRejected)` is simply calling then with no resolve reaction and only a reject reaction, `then(null, onRejected)`. `catch` is listed in Listing 5.4

The method `finally` is another function that is a simplification of a certain way to call `then`. If one wants something to happen both if a promise is rejected or resolved, one could pass the same argument to both `onFulfilled` and `onRejected`, but this is unnecessary. Instead, we can use `finally` when we want something to happen at the end of the promise chain no matter what [54]. If `onFinally` is a value, we call `then` with `onFinally` passed in both arguments [51, 26]. However, if `onFinally` is a function, we call `then` on the current promise, and if the current promise resolves, we await the resolution of `onFinally`, after which we create an already resolved promise with the value sent from the current promise. If the current promise is rejected, we still await the resolution of `onFinally`, but we throw the value the current promise was rejected with. The resulting code is listed in Listing 5.5, and the implementation is retrieved from the MDN web docs page on `finally` [26, Section 27.2.5.3].

Listing 5.6: The `flush` method.

```
flush(): void {
  if (this.fetchValue) {
    this.resolve(this.fetchValue());
  } else {
    this.precedingPromises.forEach((flushable) => flushable.
    ↪ flush());
  }
}
```

5.4 flush()

The only method of the `FlushablePromise` class that is not replicating existing promise methods meant for public use is the `flush` method, shown in Listing 5.6. This is the method we call when we want to alert the original promise that it needs to hurry up and resolve because another promise is waiting for its value. We achieve notifying the original promise of the chain by having every promise notify its preceding promise. In the constructor, we set the attribute `fetchValue` if the `onFlush` argument is provided. In the `then` method, we set the preceding promise for the new promise to be its originator. The `flush` method is pretty simple, it checks if `fetchValue` is defined, and if it is, the promise we are checking is a promise at the head of its chain that was initialized with an `onFlush` argument. If that is not the case, we call flush on each of its preceding promises.

When we call flush on a promise further down in a promise chain, it will call itself recursively until it either finds a promise with `fetchValue`, in which case it will resolve and start the chain reaction of resolving every promise in the chain, or it finds neither a `fetchValue` nor a preceding promise. In this case, we have an original promise that does not have the flush ability, and the attempt to `flush` has no effect.

5.5 resolve and reject

In Listing 5.5, we saw a static method `FlushablePromise.resolve` that we have not seen before. This is an established method in the Promise API, and its purpose is to create a resolved promise from a value. In the flushable promise class, we implement it as close to the promise specification [26, Section 27.2.4.7] as possible. It should take any value and return a promise that is resolved with that value. If the value is a promise, we simply return it. In the case of the flushable promise class, there is only one extra consideration to make: if the argument value is a flushable promise, we do not need to concern ourselves with its preceding promises, as we only return it, and do not make a new promise. The `Promise.reject(value)` method is similar, it returns a promise that has been rejected with `value`. Conforming to the specification, we create a promise, reject it with a value, and then return that promise [26, Section 27.2.4.6].

5.6 `all`, `race` and `any`

The methods `Promise.all`, `Promise.race` and `Promise.any` are all static methods belonging to the Promise API. They all take a list of promises as a parameter and return a new promise comprised of the list of promises in some way. The method `all` returns a promise that gets resolved only when all of the promises in the list of promises get resolved. The value the promise gets resolved with is a list of all of the values that each promise in the promise list got resolved with. The method `race` returns the promise in the promise list that gets settled first, and `any` returns the first fulfilled promise.

All of these methods create a new situation to consider for the preceding promises, namely several preceding promises for one promise. If we call `Promise.all([p1,p2,p3])`, and we then chain the promise result and call `flush`, we need to flush all three promises, as we do not know ahead of time what promise will resolve first. One might flush because of needing the value quicker, but it can also be used to make sure no new user input is accepted, like in the case where we press Enter on a GUI widget and it subsequently becomes inactive. In any case, we should flush all the dependent promises, and therefore we add them all as preceding promises for the promise returned by `Promise.all`

In the case of race or any, it may defeat the purpose if only one of them was created with an `onFlush` argument; if we call `Promise.race([p1,p2,p3])`, and only `p2` has a defined `onFlush`, then if we call flush on a promise in a promise chain derived from that expression, then `p2` will resolve first. Hence the programmer should consider whether using `race` or `any` in combination with flushing makes sense. In the case that we called `flush` on the result of `Promise.all` where not every promise has a defined `onFlush`, it would end up waiting for the non-flushable promises. Every promise returned by these functions will have all of the promises passed in the promise list argument as their preceding promises, and it is then up to each user of the `FlushablePromise` class to use them how they see fit.

Chapter 6

Evaluation

6.1 Testing the flushable promise

The flushable promise's purpose is to be able to send a signal upstream in a promise chain that data is needed immediately. If the original promise of the chain was created with an `onFlush` argument, we then want it to resolve with the provided flushing function. The original intended use case for this flushing functionality is the case where the resolution of a promise is blocked by a timeout due to event coalescing strategies such as debouncing.

In a chain of flushable promises where the first promise was created with a flush function, if we flush one of the chained promises , we should then expect that the original promise gets resolved with the value returned in the flush function. We also expect that this value is sent further to the rest of the promises in the chain. To test whether our implementation of the flushable promise works as intended, we can use unit tests to check what value the promises are resolved with. One framework for unit testing in JavaScript is Jest [1]. Here we are testing the JavaScript version of the `FlushablePromise` class, as Jest requires some extra work to compile TypeScript files.

In the promise chain presented in Listing 6.1, we set the `onFlush` argument of `promise_1` to return the text "flushing", and the respective chained promises to concatenate the result from the last promise with "p1" and "p2". We then create a timeout where we plan to resolve the first promise of the chain in three seconds, but we also create another timeout to flush the third promise after one second. With this we emulate the situation where we are in the middle of a timeout, waiting for the promise to get resolved, and then instead flushing whatever promise we have access to.

Listing 6.1: Creating promise chain.

```
let promise_1 = new FlushablePromise(()=>"flushing;");
let promise_2 = promise_1.then((x)=> x + " p2;",(e)=>"error
↪ ");
let promise_3 = promise_2.then((x)=> x + " p3;");
setTimeout(() => promise_1.resolve("resolving"), 3000);
setTimeout(() => promise_3.flush(), 1000);
```

To test the value of the promises after they are settled, we can use the Jest functions `test`, `expect` and `toBe`. We need `resolves` to wait for the promises to resolve, as the code is asynchronous. If `promise_3` was not able to flush the promise chain, the test would not pass, as the result passed from `promise_1` to the rest of the chain would be "resolving" and not "flushing". The tests showed in Listing 6.2 all pass.

Listing 6.2: Testing promises with Jest.

```
test('promise_1 should be flushing;', () => {
  expect(promise_1).resolves.toBe("flushing;");
});
test('promise_2 should be flushing; chain 1;', () => {
  expect(promise_2).resolves.toBe("flushing; p2;");
});
test('promise_3 should be flushing; chain 1; chain 2;', ()
↪ => {
  expect(promise_3).resolves.toBe("flushing; p2; p3;");
});
```

Note that the Jest method `resolves` only works with promises, normally giving an error message that what `resolves` is called on must be a promise, but as it probably uses the promise's `then` method, it accepts an instance of the `FlushablePromise` class here as well.

6.2 Fixing a travel planner

To evaluate the flushable promise solution in a more real-world example, we revisit the problem it is trying to solve; a GUI widget field getting input that the user is expecting it to act on immediately, while the widget itself is programmed to wait before acting on the information received for optimization purposes.

A demonstration of this problem can be found in the search for cities at the travel planner “momondo” [7]. If we type the letter “A”, and then let the search suggestions appear, “Amsterdam” is at the top of the list of search suggestions. If we then type “T” and press enter right away, we would expect the search to update with the cities whose names start with “At”. Furthermore, it would make sense for the GUI to then select the top option since we pressed Enter, which would be Atlanta. What happens instead is that Amsterdam gets chosen as our selected city. We cannot know for sure how this application has been implemented for it to act this way. However, we know that the value is not being updated right away, and it somehow stores the “old” value. This is not expected GUI behavior, and we can hypothesize that there is some waiting period implemented to alleviate the pressure of many API calls, and that they have not thought of the instance where a user presses Enter on input that still has not been processed.

To evaluate the flushable promise, we made a mock-up of a travel planner GUI to replicate the momondo travel planner, using debouncing and JavaScript promises. We first present how it could be implemented with regular promises: upon a key event from the input field, we check if there is already an unresolved promise, and if not, we create one. We subsequently chain the promise to retrieve search suggestions based on the input. This second promise then returns a list of the countries' names that match the input. The resolution of the first promise is debounced to alleviate the pressure on the hypothetical API call we would have to make to fetch the search suggestions.

When the list of country names that fit with our search is ready, it triggers the updating of the page with the search suggestions, using a third promise. If the user has already chosen a suggestion by the time we have the list of compatible countries, which would happen if they typed and pressed Enter right away, we do not need to update the page with the search suggestions. Therefore, the second promise is chained with two separate promises, the other one responsible for “autocompleting” the field with the chosen value, where this “autocompleting” action only happens if the user clicks on one of the proposed countries or presses Enter. The travel planner is illustrated in Figure 6.1

We have to make some considerations to handle the problem of delayed values upon a user's submit action. We could keep a reference to the first promise, and implement a function resolving this promise in the case that Enter or “Submit” is pressed. We would then need a way to get a hold of the correct value to resolve the promise, which would send the value further to the next promise and retrieve the search suggestions based on that. This solution is doable, but it has some drawbacks: First, the solution breaks the modularity of the application; the event handler for the “Submit” button should not have

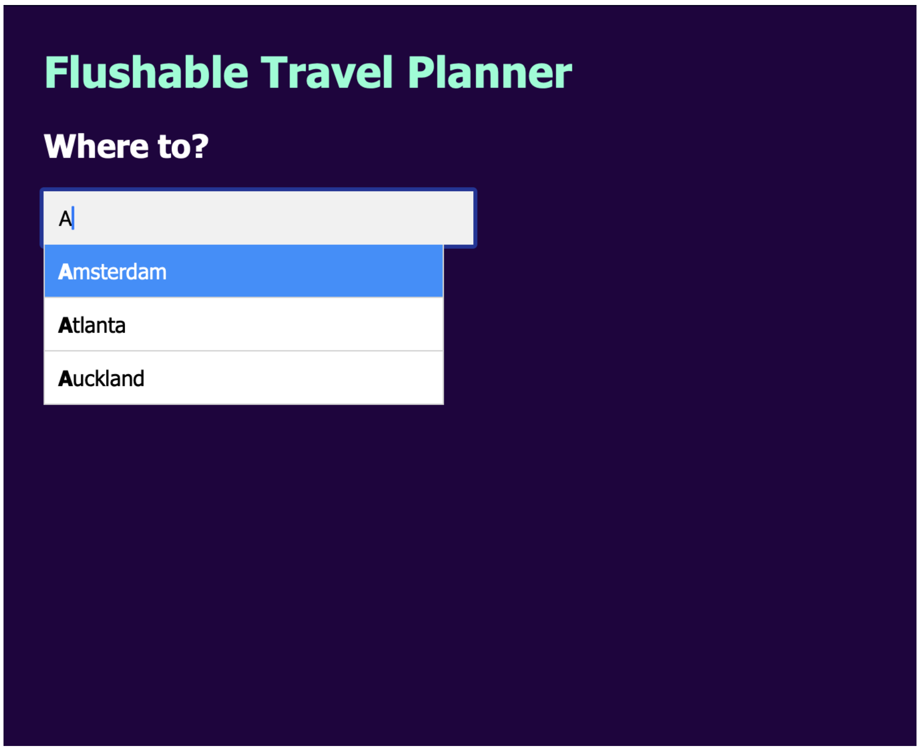


Figure 6.1: A flushable travel planner.

to “know” that the input field is debounced. It makes the code more tangled, and it is not a good abstraction for the problem. If we were to revisit this code at a later time, it would perhaps not be clear why we resolve the first promise at the “Submit” button listener. Furthermore, in GUIs with more complex dataflows, the original promise of the chain may vary depending on the current flow of data, as discussed in Section 2.3.1.

If we instead use flushable promises in our travel planner, the code remains modular. When creating the first promise, we pass a function retrieving the value from the input field as the `onFlush` argument. If the user presses Enter, we still chain the second promise with the autocompletion. With the promise returned by that operation, we call `flush`. This then bypasses the debouncing wait time and right away triggers the retrieval of the suggestions. Since the user pressed Enter, we know they want whatever is at the top, and we can then skip showing the search suggestions and just pick the first of the list.

Using the flushable promise instead of a regular promise, we first and foremost get the advantage of abstracting the problem at hand. When revisiting the code, using the flushable promise will be a more explicit way to solve the problem of delayed values than using regular promises and keeping a global reference to the first promise of the chain. Another key benefit is better modularity. The “Submit” button does not have to “know” that the input field is debounced. For the GUIs where it is especially difficult to keep a reference to the original promise, the flushable promise will be very helpful, as we only need an arbitrary promise in the chain to be able to flush the entire chain.

On the other hand, using the flushable promise is not much less work than using regular promises, as we still have to provide a function retrieving the value in both solutions. In the solution using the flushable promise, the first promise still had to be a shared variable as we had to check if it was defined at every new key event from the input field. Furthermore, since the `FlushablePromise` class is just a wrapper class at this point, it lacks some functionality that regular promises offer. One can for example not combine flushable promises with `async` and `await` for more understandable code, as `async` functions in JavaScript are translated into regular promises.

Chapter 7

Related Work

The motivation of this thesis, and the work on MDCS-based GUIs, and HotDrink specifically, is to help programmers construct high-quality user interfaces. One quality criterion of a GUI is that it behaves in an unsurprising way [32]. If the GUI is displaying some unexpected behavior, this can worsen the *gulf of evaluation* [32, 44], which refers to the amount of processing a user has to perform to determine whether or not their goal in using the GUI has been achieved. We developed the flushable promise to help programmers implement GUIs that do not exhibit defects caused by event coalescing, that is, to help them implement unsurprising GUIs.

Before implementing the flushable promise, we needed to look at existing implementations of promises and other asynchronous programming concepts, to get a better understanding of what already had been attempted in the field of asynchronous programming. In this chapter, we present other asynchronous programming concepts and how they are used in other languages. We also present some topics related to event coalescing strategies. Additionally, it was important to get a better understanding of the existing research on human-computer interaction, as the basis for needing the flushable promise revolves around improving user experience and avoiding potential surprises in the behavior of the GUI, and therefore we detail the findings of studies on response time in human-computer interaction.

7.1 Coroutines

An umbrella term for asynchronous event handling where one leaves and then revisits code is *coroutines*, although it relates more to `async/await` rather than promises. The idea of coroutines was introduced in the early 1960s and was widely explored in the following years. They lost popularity over the years, and not many years ago they were not supported by the most used programming languages [52]. As of late, they have been having a kind of "renaissance", gaining interest since the early 2000s [28]. Coroutines are generally understood as a subcontext that has its own local data that stays the same between calls, where the execution of the subcontext can be suspended and re-entered [50, 52]. The generator functions described in Section 2.2.4 are a type of coroutine; we have a function where there are local variables, and the execution context leaves the function when it encounters a `yield`, but it is possible to revisit the function until the context encounters a `return`.

How coroutines work differs greatly in different languages. Regarding the way they transfer control of the execution context, some coroutines are *symmetric* and others are *asymmetric*. The generator in JavaScript is asymmetric, as it has one function to resume execution, namely `next`, and one to suspend the execution, namely `yield`. Symmetric coroutines often have one function to suspend execution and pass it on, typically to another coroutine, and are mostly used for concurrency. Generators in JavaScript and Python are stackless, or *nonstackful*, which means we can only suspend them from the same context as they were created. If we make a second function inside a generator function in JavaScript and in the second function body we try to `yield`, we get an error. Stackful coroutines on the other hand allow suspension of the execution from within nested functions. In some languages, coroutines can only be used in certain limited contexts and in disguise, e.g., as an iterator that can only be called from within a for loop [52].

A lot of coroutines are implemented using *continuations*. A continuation is a representation of the whole control state at some point in the program execution. We can invoke the continuation at a later point, which will resume the execution of the state of the program as it was when the continuation was created [52, 42]. Continuations are used in a lot of programming languages, to for example jump out of a deeply nested function in the case of throwing an error, or in a `try-catch`-block to be able to return the state of the program to where it began if the `try`-block did not work out. Both C# and JavaScript use a *continuation-passing style* (CPS) to implement coroutines, which signifies that a continuation somehow gets passed to the function with its other parameters [28].

The way coroutines can be used for asynchronous event handling is by how they can simulate multitasking in single-threaded programs. We can do this by having a possibly blocking operation called inside a coroutine, and suspending the active routine when we encounter an operation that cannot be immediately resolved, as described in Section 2.2.5. It is hypothesized that the reason coroutines have not been prioritized earlier and have not been implemented many in general-purpose programming languages is because, with multi-threaded languages and multi-core machines, we have not needed them [52]. Multiple threads accessing the same resources present, however, a wide range of problems on their own, which coroutines could avoid. That could be why as of recently more and more programming languages are including their own versions of coroutines as they evolve.

7.1.1 Suspending functions in Kotlin

In 2017, coroutines were introduced in Kotlin [15]. Kotlin's implementation is somewhat unique, as it allows a lot of flexibility to the programmer regarding the use of coroutines. Kotlin can be compiled to JVM, JavaScript, and native binaries, and so the implementation needs to work the same way in all these platforms. The `async/await` approach has been praised for its readability for making asynchronous code look synchronous. Kotlin's designers took this kind of readability as a priority for Kotlin's coroutines. They went with a *suspending* function, which is similar to `async` function, but with the key difference that there is no need for an `await`-type keyword, as the functions marked with the `suspend` keyword in Kotlin are simply implicitly awaited [28]. This results in the asynchronous code looking no different than synchronous code.

Although the implementation of coroutines in Kotlin does not need an `await`, there are still rules to how one calls these suspending functions. A call to a suspending function cannot be made from a non-suspending function. If one wants to make an asynchronous call from within a synchronous function, one must use a *coroutine builder*: a non-suspending function where one of the parameters is a suspending lambda that is responsible for calling the coroutine [52]. These coroutine builders are not built-in but are implemented in the coroutine API, with different builders for different purposes; some are implemented to specifically block the current thread until the coroutine in question completes, while others have a specific return value.

Coroutines in Kotlin are made with continuations. When a suspendable function is invoked, it gets transformed into CPS. It then has an additional parameter representing the continuation, and its return type gets transformed into an optional any-type. A suspendable function may either suspend or return. If it is suspended, it returns a specific value `COROUTINE_SUSPENDED`. The programmer cannot manually return this value, but rather suspends the function using some established function from a `Continuation` interface. If a function returns `COROUTINE_SUSPENDED`, its caller will also return this value and so forth until the execution reaches a coroutine builder.

7.2 Asynchronous programming in other languages

Generators were introduced in JavaScript in ES6 in 2015, along with promises [25]. In 2017, `async` and `await` were introduced, but as they are a combination of generators and promises, implementations were available already in 2015 [29]. Python was early implementing generators in 2002 [52]. There have been many recent attempts at incorporating coroutines in more languages [16]. C++ recognizes all functions that contain the `await` keyword as `async` functions, and therefore does not need an `async` modifier itself [28]. Before promises in JavaScript, promise-like concepts had been introduced in various JavaScript libraries, and JQuery had a popular implementation called `deferreds`, another common word used for a placeholder value for an asynchronous operation [17].

Another term commonly used is *future*, which is used, e.g., in Rust and C++. In Rust, if one marks a function with the keyword `async` [8], it will return a future [10]. A future is a *trait*, which can be thought of as an interface [47, 10.2]. The keywords `await` [9] and `async` were introduced in 2018: one can await a future, which entails suspending the function execution until the future is completed. The future has one required method, `poll`, which attempts to resolve to future into a final value, and this is what `await` does.

7.2.1 Futures in Scala

The asynchronous concept of an object assigned to holding a value at some point in time has varying names in different programming languages. Some consider a future and a promise as two separate concepts, while others use the terms interchangeably. A future will oftentimes be seen as a placeholder for a computation that eventually will become available, while promises are a form of futures where the programmer can provide the result. Many programming languages will choose one or the other, but in Scala, both futures and promises are implemented: In the Scala documentation, futures are said to be a placeholder for an eventual computation result [39].

When one defines a future, one can state what computation it should perform, and when that computation completes. The state of a future is either `completed` or `not` `completed`. Callbacks can be used to handle a future's result asynchronously, that is, eventually after the future is `complete`. Scala is a multithreaded language, so the callback can be called on another thread than the one the future was created on. There are a few different callbacks one can use, `onComplete` is similar to `then` in JavaScript in the way that it takes a `Success` and `Failure case` for handling both possible ways a future can be settled. The `forEach` callback only handles the `Success case`. These callbacks' results cannot be chained, as they do not return a future, but Scala provides combinators, like `map` that takes a future and a mapping function for the eventual value, and returns a new future [39].

There is no `resolve` method for futures, they are described as read-only. However, promises in Scala have a future attribute, and this future can be resolved using the `success` method on the promise. One can extract the future simply by calling `p.future`, where `p` is an instance of a promise. Promises in Scala have single assignment semantics and they only be completed once. This is different from JavaScript, where if you call `resolve` on an already-resolved promise nothing happens; in Scala an exception will be thrown in such a case [39].

7.2.2 The introduction of promises

In 1988, the paper "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems" [48] defined a "new data type" they referred to as a promise. The authors Barbara Liskov and Liuba Shrira are credited to having introduced promises with this paper [52]. What motivated the development of promises was the need for a placeholder for the result of what they called a *call-stream*, a language-independent communication mechanism. They compared them to remote-procedure calls (RPCs), but whereas in RPCs a *sender* had to wait to receive the reply to a call before making another call, in call-streams the sender could make more calls before receiving a reply. In addition to this, an RPC would send a call immediately, whereas a call-stream could be sent when convenient. Promises were to be created when a call was performed, initially in a `blocked` state, and when the call returns, it switches to a `ready` state. When a promise is `ready`, it also has a value indicating whether the execution succeeded or not, and the corresponding result [48].

In Liskov's and Shrira's paper, the promise concept was attributed as an extension of the "futures" of MultiLisp [48, 40]. In MultiLisp, the future construct represented a value that was "initially undetermined" [40]: `(future x)` would return a future and begin evaluating x, and when x was finally determined, the future would be replaced by the value x was evaluated to. An operation needing the value would be suspended until the future became determined [40]. What separated promises from these futures was the error handling and pipelining of multiple promises, where a key part of the idea of promises was that the result of one call-stream could be used further in another call-stream, creating a pipeline of promises. The promise's value could also never be changed once their state switched to `ready`.

7.3 Promises/A+ specification

The Promises/A+ specification is an open standard for JavaScript promises [19]. Many have tried implementing their own versions of promises, and a consistent issue was that chaining different types of promises did not work. To rectify this issue, the Promises/A+ specification was created [24]. The specification therefore focuses on the behavior of the `then` method of promises, stating rules about the different types of arguments it should accept, what it should return, and many more details. Based on the Promises/A+ specification, a test suite has been created to check whether one's promise implementation complies with the specification [23].

7.4 `Promise.withResolvers`

Section 5.1 describes the implementation of the constructor of the flushable promise. As mentioned, we had to "extract" the methods `resolve` and `reject`, as it is useful to be able to resolve a promise with a new value at any time, and not have this aspect pre-decided upon the promise's creation. This way of saving `resolve` and `reject` to global variables is pretty common, and for this reason, a new convenience method `Promise.withResolvers` [22] was introduced to the ECMAScript Language Specification 2024 [27]. The `Promise.withResolvers` makes it much simpler to create a promise and save its `resolve` and `reject` functions for further use; see the code in Listing 7.1 and compare it to Listing 5.2. Both methods work well in the constructor of the `FlushablePromise` class, although the `Promise.withResolvers` method requires one additional line to assign all three variables to `this`.

Listing 7.1: Promise.withResolvers().

```
let { promise, resolve, reject } = Promise.withResolvers();
```

7.5 Optimizations for events in GUIs

In this thesis, when describing event coalescing strategies, debouncing and throttling have been at the center of the discussion. These are widely known and often employed to mitigate a surplus of function calls to the same function. The term debouncing is derived from *keyboard debouncing*. When we press a key on a keyboard once, two key presses might be recorded within a timeframe of one or two milliseconds. It is not possible for a human to press a key so rapidly, and so we attribute such repeated events, known as *keybounces*, to the physical or electrical properties of the keyboard. If a keybounce is recorded, we debounce the events and turn them into one keypress instead, and that is where the term debouncing stems from [41].

In database visualization GUIs, event coalescing strategies are also employed to make sure the workload does not get too big and the loading of visualizations does not lag [14]. They employ debouncing to ignore new interactions until the database management system finishes processing the queries it is currently loading, and they use throttling because the screen itself cannot refresh at a higher rate than 60-100Hz, therefore they limit the number of events handled by the application to not proceed this rate [14]. Another event coalescing strategy employed in database visualization tools is *early termination*, aborting ongoing computations if they are superseded by new interaction events [14].

ReactiveX is “An API for asynchronous programming with observable streams” [2] implemented across several languages, including Java, Swift, .NET and JavaScript. The implementations of this API employ the “Observer” pattern, the “Iterator” pattern, and functional programming to improve event handling programming [2]. The Observer pattern is a software design pattern for describing event-based control flow. This pattern defines interactions between *observers* and *subjects*, where the subjects alert the registered observers of changes [53]. The Iterator pattern is the idea of implementing an iterator for a datatype, for users of the datatype to able to traverse the elements as they please [34].

ReactiveX uses these patterns together with functional programming to treat streams of asynchronous events like one would treat an array, for example [3]. To treat many incoming calls to the same function, RxJS [11], the JavaScript implementation of ReactiveX, has numerous event filtering operators [5]. These include `debounce` and `throttle`, `sample` which emits an event in a set time interval, `distinct` that checks if an event is distinct as opposed to the previously emitted event before emitting it, and many other functions.

7.6 Human-computer interaction

In this thesis, we discuss fixing problems that result in a bad user experience, with a focus on event handling and an assumed negative effect on users when the response time to an event is too slow. In this section, we review some research on the topic of human-computer interaction and especially response times.

7.6.1 Documented effect of delay in human-computer interaction

Research on human-computer interaction indicates that delays can disrupt workflow, leading to stress, resentment, and even reduced performance in users [38, 58, 13]. When investigating delays in human-computer interaction research, we often study the system-response times (SRTs), which is the time measured from the user's input until the computer is done processing this input and able to process new input [58, 57]. A wide range of studies from the 80s and 90s have found that increased wait times can lead to frustration, perceived anxiety, and impatience [58]. These studies operated with much longer delays than we are used to in the present day, as SRTs could last up to 32 seconds [59, 18, 56], while now delays generally last between a couple of hundred milliseconds to 2.5 seconds [58].

The paper "Behavioral and emotional consequences of brief delays in human-computer interaction" [58] describes a study investigating the effect of delays in modern human-computer interactions. Participants of the study performed tasks in a computer game, under normal conditions and conditions where the game would freeze for a duration between 500 and 2800 milliseconds. Reaction times and error rates were measured to examine the behavioral performance effect of delays, and the emotional effect was documented by a questionnaire presented to the participants [58].

The researchers found that the delays significantly affected behavioral performance. The tasks performed directly after a delay showed a considerable decrease in performance, but returned to normal in the following tasks. This showed that the effect of the delay on the subsequent tasks was short-lived. The participants' general game performance was also negatively affected. As for the emotional effect of the delays, the participants liked the games with delays a lot less. The study concluded that even brief delays can have considerable negative effects on behavioral performance and the emotional state of a user [58].

7.6.2 Response times

For long it was an “established fact” that response times above 100 milliseconds were too long, but there was little research to back this up [21]. In the paper “Is 100 Milliseconds Too Fast?” [21], the authors researched whether it was true that users would notice a delay if it was over 100 milliseconds. They did this by having participants perform simple interactions with GUI elements such as typing, a menu bar and buttons, and they included delays deliberately, increasing them by five milliseconds every time the participant answered that they did not notice a delay. They found that while it varied for the different GUI tasks, the threshold for noticing delays ranged from 150 to 195 milliseconds.

Chapter 8

Discussion and Future Work

In this thesis, we have outlined a new type of JavaScript promise with the ability to ask upstream promises to resolve immediately. We have shown multiple examples where it improves the user experience and allows for a safe and modular implementation of event-handling delay strategies such as debouncing. We have also extended the λ_p calculus [49] with the flushing ability to precisely specify the semantics of this feature, to get a better understanding of how it would work if it were a functionality of a real promise, and also to be able to reason about how the flushing ability affects the properties of promises.

In Section 6.2, we saw a real-world example of flushable promises improving both the functionality and the modularity of an application; while it could be implemented without flushable promises, we would argue that flushable promises improve the readability of the code, the modularity of the different components, and lessen the possibility of errors in the application.

While the `FlushablePromise` class works for the intended purpose of alerting an earlier promise to resolve with a given value, there is still room for further improvement and work. Listed below are some possible improvements and research for future work on the flushable promise idea.

Flushable promises in HotDrink The next step in the development of flushable promises is to implement them in HotDrink, as flushable promises would be especially useful in HotDrink, and one of the key motivations for developing flushable promises was to fix the issue of waiting for a promise "upstream" to resolve, without knowing which promise to resolve. HotDrink uses promises for "variable activations", which are representations of variables' values after each user interaction. There was unfortunately not enough time to implement flushable promises in HotDrink, as HotDrink has a large code base and uses promises in a complicated manner. Furthermore, HotDrink uses `async` and `await` to chain operations, which only work with regular promises.

async/await In Section 6.2, the downside to the `FlushablePromise` class that it is not being compatible with `async` and `await` was mentioned. A further goal is to implement `async` and `await` that work for objects of the `FlushablePromise` class. As discussed in Section 2.2.5, `await` is very similar to `yield`, thus a generator would be a good place to start for the implementation of `await` for flushable promises. Babel, a well-known transpiler for introducing new syntax in JavaScript, may also be employed to transpile statements with `async` and `await` into flushable promise method calls.

Promises/A+ specification The Promise/A+ specification is a specification of the `then` method of promises [19], with the aspiration of providing a framework that one's own implementation of promises should adhere to if we would want to chain separate types of promises, as discussed in Section 7.3. The `FlushablePromise` class is not an implementation of a new promise, but rather a wrapper class, as the promise functionality itself works fine for its purpose. However, the rules discussed in Section 4 are made with the idea that it is a regular promise with the added functionality of flushing, and it would be interesting to attempt to implement a flushable promise from scratch, using the Promises/A+ specification and the λ_{fp} calculus as a basis.

Expansion of λ_{fp} As mentioned in Section 4.3, we could neither define the rule for the static promise method `Promise.all`, nor show an implementation of it with our existing rules. It would be interesting to look at this some more, as well as other promise methods such as `Promise.race` and `Promise.any`. Especially in the attempt to implement flushable promises as its own type of promise and not a promise wrapper class like discussed above, it would be useful to formalize all aspects of the functionality, and derive the implementation from the formalization.

Usability study In this thesis we have made some claims of why flushable promises would work better in certain situations than regular promises, including concerning the understandability of the implementation. We have not conducted any user tests to check whether this is true, and it would be useful to be able to see how understandable the concept of flushable promises are for developers, and if their use is perceived as intuitive or not.

A potential ECMAScript proposal Although in this thesis we have mostly discussed flushable promises' usefulness in libraries such as HotDrink, which have complicated networks of dependent promises, it is not unthinkable that there are other use areas where their functionality is useful as well. To this end, we think this thesis might lay the groundwork and may be a contender for an addition to the ECMA-262 language standard [27].

Bibliography

[1] Jest 29.7. https://jestjs.io. [Online; accessed May 2024].

[2] ReactiveX. http://reactivex.io/, . [Online; accessed May 2024].

[3] ReactiveX. https://reactivex.io/intro.html, . [Online; accessed May 2024].

[4] Difference between async/await and ES6 yield with generators. https://stackoverflow.com/questions/36196608/difference-between-async-await-and-es6-yield-with-generators, 2019. [Online; accessed May 2024].

[5] Filtering. https://www.learnrxjs.io/learn-rxjs/operators/filtering, 2020. [Online; accessed May 2024].

[6] Boliglånskalkulator. https://www.dnb.no/lan/kalkulator/boliglan, 2024. [Online; accessed april 2024].

[7] Billige flybilletter - Søk og sammenlign flyreiser online | momondo, May 2024. **URL:** https://www.momondo.no. [Online; accessed May 2024].

[8] async - Rust, May 2024. **URL:** https://doc.rust-lang.org/std/keyword.async.html. [Online; accessed May 2024].

[9] await - Rust, May 2024. **URL:** https://doc.rust-lang.org/std/keyword.await.html. [Online; accessed May 2024].

[10] Future in std::future - Rust, May 2024. **URL:** https://doc.rust-lang.org/std/future/trait.Future.html. [Online; accessed May 2024].

[11] RxJS. https://github.com/ReactiveX/rxjs, 2024. [Online; accessed May 2024].

[12] Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. Finding broken promises in asynchronous JavaScript programs. *Proc. ACM Program. Lang.*, 2

(OOPSLA), oct 2018. doi: 10.1145/3276532.
URL: `https://doi.org/10.1145/3276532`.

[13] Raymond E. Barber and Henry C. Lucas. System response time operator productivity, and job satisfaction. *Commun. ACM*, 26(11):972–986, nov 1983. ISSN 0001-0782. doi: 10.1145/182.358464.
URL: `https://doi.org/10.1145/182.358464`.

[14] Leilani Battle, Philipp Eichmann, Marco Angelini, Tiziana Catarci, Giuseppe Santucci, Yukun Zheng, Carsten Binnig, Jean-Daniel Fekete, and Dominik Moritz. Database Benchmarking for Supporting Real-Time Interactive Querying of Large Data. pages 1571–1587, 06 2020. doi: 10.1145/3318464.3389732.

[15] Roman Belov. Kotlin 1.1 Released With JavaScript Support, Coroutines, and More | The Kotlin Blog, May 2017.
URL: `https://blog.jetbrains.com/kotlin/2017/03/kotlin-1-1`. [Online; accessed May 2024].

[16] Bruce Belson, Jason Holdsworth, Wei Xiang, and Bronson Philippa. A Survey of Asynchronous Programming Using Coroutines in the Internet of Things and Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 18(3), jun 2019. ISSN 1539-9087. doi: 10.1145/3319618.
URL: `https://doi.org/10.1145/3319618`.

[17] Trevor Burnham. *Async JavaScript: Build More Responsive Apps with Less Code.* Pragmatic Bookshelf, 2012. ISBN 9781937785277.

[18] T. W. Butler. Computer response time and user performance during data entry. *AT&T Bell Laboratories Technical Journal*, 63(6):1007–1018, 1984. doi: 10.1002/j.1538-7305.1984.tb00110.x.

[19] Brian Cavalier and Domenic Denicola. Promises/A+ Promise Specification. `https://promisesaplus.com`, 2014. [Online; accessed May 2024].

[20] B. Cherny. *Programming TypeScript: Making Your JavaScript Applications Scale.* O'Reilly Media, 2019. ISBN 9781492037620.
URL: `https://books.google.no/books?id=Y-mUDwAAQBAJ`.

[21] James R. Dabrowski and Ethan V. Munson. Is 100 Milliseconds Too Fast? In *CHI '01 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '01, page 317–318, New York, NY, USA, 2001. Association for Computing Machinery. ISBN

1581133405. doi: 10.1145/634067.634255.
URL: https://doi.org/10.1145/634067.634255.

[22] Chris de Almeida. proposal-promise-with-resolvers. https://github.com/tc39/proposal-promise-with-resolvers, 2024. [Online; accessed May 2024].

[23] Domenic Denicola. Promises/A+ Compliance Test Suite. https://github.com/promises-aplus/promises-tests, 2017. [Online; accessed May 2024].

[24] Romario Diaz. Promises/A+ and thenables. https://medium.com/@RomarioDiaz25/promises-a-and-thenables-664073939cf3, 2023. [Online; accessed May 2024].

[25] Ecma. ECMAScript® 2015 Language Specification. = https://262.ecma-international.org/6.0/,, June 2015.

[26] Ecma. ECMAScript 2023 Language Specification, 2023.
URL: https://tc39.es/ecma262/2023/. [Online; accessed May 2024].

[27] Ecma. ECMAScript 2025 Language Specification . = https://tc39.es/ecma262/multipage/, 2024. [Online; accessed May 2024].

[28] Roman Elizarov, Mikhail Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. Onward! 2021, page 68–84, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450391108. doi: 10.1145/3486607.3486751.
URL: https://doi.org/10.1145/3486607.3486751.

[29] Ian Elliot. *JavaScript ASYNC.* I/O Press, November 2017.

[30] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex.* Mit Press, 2009.

[31] Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. *SIGPLAN Not.*, 51(3):121–130, oct 2015. ISSN 0362-1340. doi: 10.1145/2936314.2814207.
URL: https://doi.org/10.1145/2936314.2814207.

[32] John Freeman, Jaakko Järvi, Wonseok Kim, Mat Marcus, and Sean Parent. Helping programmers help users. *SIGPLAN Not.*, 47(3):177–184, oct 2011. ISSN 0362-1340. doi: 10.1145/2189751.2047892.
URL: https://doi.org/10.1145/2189751.2047892.

[33] John Freeman, Jaakko Järvi, and Gabriel Foust. Hotdrink: a library for web user interfaces. *SIGPLAN Not.*, 48(3):80–83, sep 2012. ISSN 0362-1340. doi: 10.1145/2480361.2371413.
URL: `https://doi.org/10.1145/2480361.2371413`.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612.
URL: `http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1`.

[35] Raju Gandhi. *JavaScript next: Your complete guide to the new features introduced in JavaScript, starting from ES6 to ES9.* APRESS, New York, NY, October 2019. ISBN 978-1-4842-5394-6. doi: 10.1007/978-1-4842-5394-6_14.
URL: `https://doi.org/10.1007/978-1-4842-5394-6_14`.

[36] Jack Ganssle. A Guide to Debouncing. Sept 2004.

[37] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. *The Essence of JavaScript*, page 126–150. Springer Berlin Heidelberg, 2010. ISBN 9783642141072. doi: 10.1007/978-3-642-14107-2_7.
URL: `http://dx.doi.org/10.1007/978-3-642-14107-2_7`.

[38] Jan L. Guynes. Impact of system response time on state anxiety. *Commun. ACM*, 31(3):342–347, mar 1988. ISSN 0001-0782. doi: 10.1145/42392.42402.
URL: `https://doi.org/10.1145/42392.42402`.

[39] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanoic. Futures and Promises, May 2024.
URL: `https://docs.scala-lang.org/overviews/core/futures.html`. [Online; accessed May 2024].

[40] Robert H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, 1985.
URL: `https://api.semanticscholar.org/CorpusID:1285424`.

[41] S. Hansen and T. V. Fossum. *Event Based Programming.* May 2010.

[42] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3):143–153, 1986. ISSN 0096-0551. doi: https://doi.org/10.1016/0096-0551(86)90007-X.
URL: `https://www.sciencedirect.com/science/article/pii/009605518690007X`.

[43] Muzzamil Hussain. *Mastering JavaScript Promises.* Packt Publishing, 2015. ISBN 978-1-78398-550-0.

[44] Edwin L Hutchins, James D Hollan, and Donald A Norman. Direct manipulation interfaces. *Human–Computer Interaction*, 1(4):311–338, 1985.
URL: `https://worrydream.com/refs/Hutchins_1985_-_Direct_Manipulation_Interfaces.pdf`.

[45] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08, page 89–98, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582672. doi: 10.1145/1449913.1449927.
URL: `https://doi.org/10.1145/1449913.1449927`.

[46] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, mar 2004. ISSN 0360-0300. doi: 10.1145/1013208.1013209.
URL: `https://doi.org/10.1145/1013208.1013209`.

[47] Steve Klabnik and Carol Nichols. *The Rust Programming Language.* February 2024.
URL: `https://doc.rust-lang.org/book/ch10-02-traits.html`. [Online; accessed May 2024].

[48] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, jun 1988. ISSN 0362-1340. doi: 10.1145/960116.54016.
URL: `https://doi.org/10.1145/960116.54016`.

[49] Magnus Madsen, Ondřej Lhoták, and Frank Tip. A Model for Reasoning about JavaScript Promises. 1(OOPSLA), Oct 2017. doi: 10.1145/3133910.
URL: `https://doi.org/10.1145/3133910`.

[50] Christopher D Marlin. *Coroutines: a programming methodology, a language design and an implementation.* Number 95. Springer Science & Business Media, 1980.

[51] MDN Web Docs. Promise.prototype.finally() - JavaScript — MDN. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/finally`, 2024. [Online; accessed May 2024].

[52] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2), feb 2009. ISSN 0164-0925. doi: 10.1145/

1462166.1462167.
URL: `https://doi.org/10.1145/1462166.1462167`.

[53] Gleb Naumovich. Using the observer design pattern for implementation of data flow analyses. *SIGSOFT Softw. Eng. Notes*, 28(1):61–68, nov 2002. ISSN 0163-5948. doi: 10.1145/634636.586107.
URL: `https://doi.org/10.1145/634636.586107`.

[54] Daniel Parker. *JavaScript with Promises.* O'Reilly Media, Sebastopol, CA, October 2014.

[55] John Resig, Bear Bibeault, and Josip Maras. *Secrets of the JavaScript Ninja.* Manning Publications Co., USA, 2nd edition, 2016. ISBN 1617292850.

[56] Florian Schaefer. The Effect of System Response Times on Temporal Predictability of Work Flow in Human-Computer Interaction. *Human Performance*, 3(3):173–186, 1990. doi: 10.1207/s15327043hup0303_3.
URL: `https://doi.org/10.1207/s15327043hup0303_3`.

[57] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3):265–285, sep 1984. ISSN 0360-0300. doi: 10.1145/2514.2517.
URL: `https://doi.org/10.1145/2514.2517`.

[58] André J. Szameitat, Jan Rummel, Diana P. Szameitat, and Annette Sterr. Behavioral and emotional consequences of brief delays in human–computer interaction. *International Journal of Human-Computer Studies*, 67(7):561–570, 2009. ISSN 1071-5819. doi: https://doi.org/10.1016/j.ijhcs.2009.02.004.
URL: `https://www.sciencedirect.com/science/article/pii/S1071581909000329`.

[59] Stuart Martin Weiss, George Boggs, Mark Lehto, Sogand Shodja, and David J. Martin. Computer System Response Time and Psychophysiological Stress II. *Proceedings of the Human Factors Society Annual Meeting*, 26(8):698–702, 1982. doi: 10.1177/154193128202600805.
URL: `https://doi.org/10.1177/154193128202600805`.

[60] WHATWG. HTML Living Standard. `https://html.spec.whatwg.org/`, 2024.

[61] WHATWG. Fetch Living Standard. `https://fetch.spec.whatwg.org`, 2024.

Appendix A

Remaining Promise λ Rules

Below are the remaining rules of the λ_{fp} calculus not presented in Chapter 4. The rules [E-OnReject-Rejected], [E-Reject-Pending], [E-Reject-Settled], and [E-Link-Rejected] are not presented, as they are conceptually similar to other rules.

E-Context

$$\frac{e \hookrightarrow e'}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ e\ ] \rangle \rightarrow \langle \sigma', \psi', f', r', \pi', \varphi', \delta', E[\ e'\ ] \rangle}$$

E-OnResolve-Fulfilled

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) = \mathsf{F}(v) \qquad a' \in \mathit{Addr} \\ a' \notin \mathsf{dom}(\sigma) \qquad \psi' = \psi[\ a' \mapsto \mathsf{F}(v)\ ] \qquad \sigma' = \sigma[\ a' \mapsto \{\}\ ] \\ f' = f[\ a' \mapsto \mathit{Nil}\ ] \qquad r' = r[\ a' \mapsto \mathit{Nil}\ ] \qquad \pi' = \pi ::: (\mathsf{F}(v), \lambda, a') \end{array}}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a.\mathtt{onResolve}(\lambda)\ ] \rangle \rightarrow \langle \sigma', \psi', f', r', \pi', \varphi, \delta, E[\ a'\ ] \rangle}$$

E-Resolve-Settled

$$\frac{a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) \in \{\mathsf{F}(v'), \mathsf{R}(v')\}}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a.\mathtt{resolve}(v)\ ] \rangle \rightarrow \langle \sigma, \psi, f, r, \pi, \varphi, \delta, , E[\ \mathsf{undef}\ ] \rangle}$$

E-Resolve-Pending

$$\frac{\begin{array}{c} a \in \mathit{Addr} \qquad a \in \mathsf{dom}(\sigma) \qquad \psi(a) = \mathsf{P} \\ f(a) = (\lambda_1, a_1) \cdots (\lambda_n, a_n) \qquad \pi' = \pi ::: (\mathsf{F}(v), \lambda_1, a_1) \cdots (\mathsf{F}(v), \lambda_n, a_n) \\ \psi' = \psi[a \mapsto \mathsf{F}(v)] \qquad f' = f[\ a' \mapsto \mathit{Nil}\ ] \qquad r' = r[\ a' \mapsto \mathit{Nil}\ ] \end{array}}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a.\mathtt{resolve}(v)\ ] \rangle \rightarrow \langle \sigma, \psi', f', r', \pi', \varphi, \delta, , E[\ \mathsf{undef}\ ] \rangle}$$

E-Link-Fulfilled

$$\frac{a_1 \in \mathit{Addr} \qquad a_1 \in \mathsf{dom}(\sigma) \qquad a_2 \in \mathsf{Addr} \qquad a_2 \in \mathsf{dom}(\sigma) \qquad \psi(a_1) = \mathsf{F}(v) \qquad \pi' = \pi ::: (\mathsf{F}(v), \text{default}, a_2)}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta, E[\ a_1.\mathtt{link}(a_2)\ ] \rangle \rightarrow \langle \sigma, \psi, f, r, \pi', \varphi, \delta, E[\ \mathsf{undef}\ ] \rangle}$$

E-Loop-Fulfilled-Lambda

$$\frac{\pi = (\mathsf{F}(v'), \lambda_r, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta \rangle \rightarrow \langle \sigma, \psi, f, r, \pi', \varphi, \delta, a.\mathtt{resolve}(\lambda_r(v')) \rangle}$$

E-Loop-Rejected-Lambda

$$\frac{\pi = (\mathsf{R}(v'), \lambda_r, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta \rangle \rightarrow \langle \sigma, \psi, f, r, \pi', \varphi, \delta, a.\mathtt{resolve}(\lambda_r(v')) \rangle}$$

E-Loop-Fulfilled-Default

$$\frac{\pi = (\mathsf{F}(v'), \mathit{default}, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta \rangle \rightarrow \langle \sigma, \psi, f, r, \pi', \varphi, \delta, a.\mathtt{resolve}(v') \rangle}$$

E-Loop-Rejected-Default

$$\frac{\pi = (\mathsf{R}(v'), \mathit{default}, a) :: \pi'}{\langle \sigma, \psi, f, r, \pi, \varphi, \delta \rangle \rightarrow \langle \sigma, \psi, f, r, \pi', \varphi, \delta, a.\mathtt{resolve}(v') \rangle}$$