Implementing a browser-based extension of JavaScript that supports Oz-style concurrent constraint programming

Anubhav, A0167968A

November 4, 2020

Contents

1 Introduction

1.1 Motivation for Project

Concurrent programming is an essential tool for building software. However, it can be difficult to ensure the correctness of concurrent programs, because their behaviour depends heavily on the nondeterministic scheduling of instructions. Extra effort has to be expended to ensure that a program will be correct, regardless of the ordering of instructions. Similarly, it is challenging to reproduce bugs that only surface under a specific interleaving of the instructions.

Oz is a general purpose programming language that offers a panacea to these woes. Oz's programming model enables *declarative concurrency*. Declarative concurrency refers to concurrent programming that is fully deterministic, regardless of the scheduling of the concurrent components in the program (Van Roy & Haridi, 2004).

Oz also provides a sophisticated constraint programming system. Constraints can be specified and combined using unification, and then evaluated using a powerful constraint solving system that infers additional constraints at runtime.

This project aims to create an extension of JavaScript that supports both Oz-style declarative concurrency and a simplified version of Oz's constraint programming. Programmers will be able to write code directly in this extension of JavaScript. Such an extension of JavaScript could benefit both casual and professional programmers. Casual programmers will be able to broaden their exposure by learning about declarative concurrency in the comfort of their browser. Professional programmers could adopt declarative concurrency in their web applications.

In this report, I will first briefly explain the semantics of declarative concurrency and unification in Oz. I will then review existing work that is relevant to my project. Finally, I will discuss and evaluate my current language implementation. My plans for the next leg of this project can be found in §4.2 and §5.

1.2 Oz semantics: Declarative concurrency and Unification

Oz is able to provide deterministic concurrency because variables in Oz satisfy three special properties:

1. Every variable is unbound (i.e. not representing any value - *undetermined*) when it is first declared.

2. Once a variable is bound to a value, this binding is immutable. The variable cannot be subsequently bound to a new value.

3. If a thread attempts to use an unbound variable in an operation, that thread will get suspended, until the variable is bound to a value.

These three properties guarantee that we can write deterministic concurrent programs that have an arbitrary numbers of threads operating on shared data structures. The following code snippet gives an example of declarative concurrency in an Oz program that launches two threads. The first thread depends on values produced by the second thread. Hence, the first thread automatically suspends until `F` and `G` have been produced.

```
local F G in

   thread
      {Browse F + G} % prints 15 after two seconds
   end

   thread
      {Delay 2000} % suspend for two seconds
      F = 5
      G = 10
   end

end
```

We see that the Oz language's semantics allow operations to be deferred until their operands are available. This behaviour is called *dataflow execution* (Van Roy & Haridi, 2004).

In the example above, we saw the `=` operator being used to bind Oz variables. This operator represents unification in Oz. In Oz, every value is considered to be a rooted directed graph, whose nodes either hold primitive values or point to other nodes. The unification takes two operands, and it adds edges to their graphs until both operands are structurally equal. Hence, the unification below will turn both lists structually equal.

Oz's variables are commonly called *logic variables* to emphasise the presence of the unification operation. I will also use this name to describe Oz-style variables in this report.

1.3 Project Objectives

To support Oz-style declarative concurrency and a simple constraint programming model, we will need to implement logic variables and a threads' abstraction that can be used conveniently in our JavaScript extension.

Below is a list of specific goals for this project:

1. The JavaScript extension should be executable in modern web browsers, without requiring users to install software onto their computers.
2. Programmers should be able to create, unify and use logic variables. In addition, basic pattern matching on logic variables should be possible.
3. Programmers should be able to launch threads and manage running threads.

Currently, this project implements logic variables using promise-like JavaScript objects. Threads are implemented using a transpiler that transforms threads in the JavaScript extension to asynchronous JavaScript functions in the native JavaScript code. The transpiler also adds suspension points into the native code, to enable context switching.

2 Literature Review

In this section, I will review work that is relevant for this project. In §2.1, I will review existing JavaScript support for computing with undetermined values. I will show the similarity between JavaScript promises and Oz-style logic variables.

In §2.2, I will review support for threads in both standard JavaScript and community projects.

In §2.3, I will review JavaScript support for pattern matching.

In §2.4, I will review the implementation of the Oz virtual machine. I will explain why I did not adopt a virtual machine based implementation in my current language implementation.

2.1 Existing JavaScript support for computing with undetermined values

JavaScript does not provide constructs that exactly match the semantics of Oz's logic variables. However, it does support a related concept; JavaScript allows programmers to specify computations on values that will be *available eventually*.

This aspect of JavaScript is heavily used in web applications, because such applications do need to perform work with values received asynchronously from users or servers.

Callback functions
Callbacks[1] provide one mechanism for specifying computations on values that will be *available eventually*. Callbacks functions are paired with specific browser events, and they are executed when those events occur. By pairing callback functions with the right browser events, we can compute on values that are asynchronously received from users or web servers.

Promises
Callback functions allow JavaScript programmers to compute with asynchronously received values. However, as mentioned by Simpson (2015), callbacks lend themselves to highly unmaintantable coding patterns. Promises were added to JavaScript in 2015. They also allow computing with asynchronously received values, but with cleaner coding patterns than callbacks.

A promise represents a value that will be eventually available. To create a promise, we pass a function that will eventually produce a value, into the `Promise` constructor. The example below shows how we could create a promise that eventually produces the HTTP response received from a server.

```
const httpResponse = new Promise((resolve, reject) => {
    const url = 'https://test-endpoint.com/userdata';
    ... // code for performing HTTP request
});
```

The Promise construct provides a `.then()` method that is used to specify computations that should be performed once a promise's value is availabe. Suppose that the promise `httpResponse` eventually resolves to the the value `a` representing a HTTP response. The code `httpReqest.then(e1)` serves to attach a function `e1` to the `httpResponse` promise, and it **returns a new promise** whose value will be `e1(a)`.

[1]Depending on the usage, callbacks may not be the same as continuations. For example, callbacks are not always applied at the end of a computation.

We can hence setup a chain of `.then()` function calls that represent a sequence of computations that should be executed once the `httpRequest` promise resolves to a determined value. The example below shows how we could perform some business logc with the `httpResponse` promise. In this case, we are updating the user interface's title with the number of users available in our database.

```
httpResponse.then(response => response.json())
            .then(data => data.numOfUsers)
            .then(numOfUsers => {
                    title.text = `${numOfUsers} profiles in our database!`)
                  });
```

Asynchronous JavaScript functions

In 2016, the JavaScript language obtained asynchronous functions. These functions do not add any new functionality. They are simply syntactic sugar for promises.

The semantics of asynchronous functions can be described as follows:

1. Applying an asynchronous function gives us a promise to the function's result.
2. Within the body of an asynchronous function, we can suspend the function's execution by applying the `await` keyword to a promise `e`. The function will suspend until `e` provides a value.

The example below shows that asynchronous functions give us a promise when they are applied. Specfically, note that calling the `makeHttpRequest` asynchronous function does not block the JavaScript runtime. The runtime simply obtains a promise, and continues executing past the function call. We can chain `.then()` methods on promise as usual.

```
async function makeHttpRequest(url) {
    const response; // variable to hold HTTP response

    ...
    ...
    ... // some code that performs a HTTP request

    return response;
}

const httpResponse = makeHttpRequest(`https://test-endpoint.com/userdata`);
httpResponse.then(response => response.json())
               .then(data => data.numOfUsers)
               .then(numOfUsers => {
                  title.text = `${numOfUsers} profiles in our database!`)});
```

The example below shows how we could condense the `.then()` chain above into a single `async` function. Internally, the `getNumOfUsers` function will suspend till the HTTP response is received, because we have applied `await` to the promise returned from `makeHttpRequest`. Externally, calling `getNumOfUsers` will not block the JavaScript runtime.

```
async function makeHttpRequest(url) {
    const response; // variable to hold HTTP response

    ...
    ...
    ... // some code that makes a HTTP request

    return response;
}

async function getNumOfUsers() {
    const response = await makeHttpRequest('https://test-endpoint.com/userdata');
    const responseJson = response.json();
    const data = responseJson.numOfUsers;
    title.text = '${numOfUsers} profiles in our database!';
}

getNumOfUsers(); // trigger the HTTP request and subsequent computation
```

In summary, `async` functions provide syntactic sugar for creating promises and the `await` keyword provides syntactic sugar for the deferred computation sequences specified in `.then()` chains. These features are relevant for this project because they will be used in the implementation of logic variables.

Comparing JavaScript constructs with Oz logic variables

Both callbacks and promises are similar to Oz's logic variables because all of these constructs enable *dataflow execution*. Recall from §1.2 that dataflow execution means that operations are deferred until their operands are available. This is possible with callbacks; we can place our operations inside callbacks, and then pair these callbacks with events that are fired whenever the operands are available. Dataflow is also clearly possible with promises; we can place our operations inside `.then()` calls attached to a promise of our operands.

However, the dataflow execution possible with promises and callbacks is much more restrictive, compared with the dataflow techniques possible with logic variables. For example, it is not possible to bind a promise to a value after instantiating the promise. The value of a promise can only be provided by the function passed into the `Promise` constructor. Conversely, Oz's logic variables can be declared and bound at different points in the program. This limitation also prevents us from implementing unification on standard JavaScript promises, since unification often involves binding an undetermined variable to a determined value.

2.2 Existing projects that provide the threads' abstraction in JavaScript runtimes

The JavaScript runtime in web browsers does not support the concept of concurrent threads that can be pre-emptively suspended at any instruction. Instead, it achieves a coarse-grained form of concurrency using an event-driven system. The runtime sequentially executes units of JavaScript code associated with the events occuring on a web page (e.g. mouse clicks, HTTP responses, timer events). Event handlers can hence execute in an interleaved fashion, giving

users an appearance of concurrency.
In this section, I will discuss some recent projects aimed at achieving more fine-grained concurrency (and even parallelism) in JavaScript runtimes.

Doppio

Doppio is a runtime environment written in Javascript. It is intended to be a target runtime for language implementors who want to port their programming language to browsers. Most traditional languages expect the runtime environment to support multiple threads of execution, perhaps even with pre-emptive switching.

However, as mentioned above, browser-based JavaScript runs on a single thread, with an event-driven system used to achieve coarse-grained concurrency. The Doppio project hence implemented a threads' abstraction in their runtime. This is achieved by setting two requirements for language implementations that target Doppio. First, each implementation must store the callstacks of its threads explicitly in JavaScript objects. Second, each implementation must periodically check whether it should suspend and give up control to the Doppio scheduler. For example, in an implementation of the Java Virtual Machine that targets the Doppio runtime, one suspension check is done on every function call (Vilk & Berger, 2014).

The Doppio scheduler maintains an array of call stacks - one for every thread. When a language implementation chooses to suspend one of its running threads, the scheduler will select one of the other stacks in its array for resumption.

Comparing Doppio with this project

Doppio is highly relevant to this project, because it has implemented a threads' abstraction in browser-based JavaScript. To achieve mulithreading in my project, I have adopted a similar strategy as Doppio; my transpiler inserts suspension points into the generated JavaScript code.

However, my current implementation also differs significantly from Doppio. Instead of managing threads' call stacks explicitly, threads in my language implementation can simply use the call stack provided by the JavaScript runtime in the browser. This is because my current design implements threads as asynchronous JavaScript functions.

Web workers

Modern web browsers provide parallelisation constructs known as *web workers*. These workers are able to execute JavaScript code in a separate thread, while the browser's main thread continues processing events associated with the web page. Web applications can benefit from web workers by delegating computationally intensive tasks to them (Costa, 2017).

In 2016, a shared memory primitive, `SharedArrayBuffer`, was added to JavaScript. This primitive allows the main thread to share an array of bytes with the workers. Support for performing atomic operations on this shared memory has also been added to the language. This includes the ability for workers threads to sleep until a specific index of the shared array has been updated.

With this shared memory primitive, web workers could be used to implement the threads' abstraction for this project. This approach is likely to be more complex than the current design, because the program's shared data will need to be managed explicitly using `SharedArrayBuffer`.

2.3 Existing JavaScript support for pattern-matching

Oz has a powerful constraint solving and pattern matching system. JavaScript does not provide these features, but there are often community projects that extend JavaScript's capabilities.

For example, the `z` pattern matching library allows programmers to test the structure of an expression `v1`, and then execute code conditionally based on these tests. The code snippet below shows how we can match against both the type and the value of an expression. [2]

```
const { matches } = require('z')

const result = matches(1)(
  (x = 2)      => 'number 2 is the best!!!',
  (x = Number) => `number ${x} is not that good`,
  (x = Date)   => 'blaa.. dates are awful!'
);
```

This library implements pattern matching by using a parser to extract the constraints from each pattern. The library does achieve its purpose, but it has effectively defined a mini-language for pattern matching. For this project, I have opted for simpler pattern matching that can be understood with JavaScript's existing semantics. My current idea is to implement pattern matching by sequentially testing a list of predicates, and evaluating the expression paired with the first true predicate.

2.4 Mozart - the Oz virtual machine

The language Oz itself runs on a virtual machine named Mozart, which is implemented in C++. This virtual machine is register-based and single threaded. It provides pre-emptive, user level threads that are managed by a scheduler. Oz code is first compiled into a simpler subset of Oz, before being compiled into Mozart machine instructions. These instructions are then evaluated by Mozart.

Mozart provides a reference implementation for the Oz language. For my work, I have referred to the unification algorithm used in Mozart and described by Mehl (1999). However, I chose not to pursue a virtual machine based language implementation. I have instead adopted a transpiler based approach.

There are two reasons for this choice. Firstly, a virtual machine approach may require additional effort towards optimisation. We would have to optimise the virtual machine, the compilation process and the machine instruction set. Conversely, in a transpiler based approach, we would only need to optimise the transpilation process and the JavaScript emitted by the transpiler. The JavaScript execution would not need additional optimisation, because we can enjoy the optimisations built into modern web browsers.

Secondly, a transpiler based approach would make it easier to share programs written in this JavaScript extension. It would suffice to share the code generated by the transpiler, because any modern browser could execute it. Conversely, a virtual machine based approach would require users to share the virtual machine program together with their code.

[2] Code snippet has been taken from the examples on this library's GitHub page

3 Current design and implementation

In this section, I will describe the progress made thus far, in the implementation of this JavaScript extension. In §3.1, I will explain how I have implemented logic variables using promise-like objects in JavaScript. In §3.2, I will explain how I have created a transpiler that implements threads as asynchronous functions.

3.1 Logic variables

As we saw in §2.1, promises are only able to provide a restricted form of dataflow execution because they cannot be bound to values after instantiation. We cannot implement unification on promises for the same reason.

To implement the full semantics of logic variables (unrestricted dataflow and unification), I have implemented a special type of promise which can be bound to values after instantiation.

This special promise keeps the dataflow behaviour of normal promises; it can suspend the execution of `async` functions. In the JavaScript ecosystem, objects that can be used like promises are called *thenables* or *promise-likes*. I will use the term *promise-like* for simplicity. Below is the JavaScript class definition of this promise-like. [3]

```
class LogicVariable {
    constructor(value=undefined) {
        if (value === undefined) {
            this.value = undefined;
            this.state = TRANSIENT;
        } else {
            this.value = value;
            this.state = DETERMINED;
        }
    }

    then(onFulfilled) {
      if (this.isTransient()) {
        setTimeout(() => this.then(onFulfilled), 0);
      } else if (this.isDetermined()) {
        onFulfilled(this.value);
      } else if (this.isReference()) {
        this.value.then(onFulfilled);
      }
    }

    unify(other) {
      ... // follows the algorithm described by Mehl(1999)
    }
}
```

[3] Some trivial class methods have been omitted for brevity

This promise-like `LogicVariable` has three possible states: [4]

1. Transient: In this state, the `LogicVariable` has not been bound to a value. It is undetermined.
2. Determined: This means the `LogicVariable` has been bound to a value.
3. Reference: This means the `LogicVariable` has been bound to another `LogicVariable` (which itself may be in any of the three states)

A `LogicVariable` can be created by simply invoking the constructor. If a value is provided to the constructor, the `LogicVariable` becomes immediately determined. Otherwise, it remains transient.

```
const integer = new LogicVariable(10); // a determined LogicVariable with value 10
const unknown = new LogicVariable(); // a transient LogicVariable
```

The code snippet below shows that we can unify the logic variables with the `unify` method. The unification algorithm follows Mehl(1999)'s description. Its full implementation can be found in the Appendix.

```
const integer = new LogicVariable(10); // a determined LogicVariable with value 10
const unknown = new LogicVariable(); // a transient LogicVariable
unknown.unify(integer); // the two LogicVariables are now structurally equal
```

The dataflow semantics expected from Oz-like logic variables are implemented by the `then`. In JavaScript's asynchronous functions, the `await` keyword can be applied to any object that has a `then` method. Our `LogicVariable` can hence be used with `await`.

Specifically, when we evaluate the expression `await e1`, the JavaScript runtime invokes `e1.then(onFulfilled)`, where `onFulfilled` is a callback expected to eventually receive the value of `e1`. In the `LogicVariable`'s `then` method, the callback is only applied if the `LogicVariable` **is determined**. If the `LogicVariable` is transient, the `.then` method marks itself to be invoked again, as soon as possible. This causes the evaluation of `await e1` to suspend the `async` function, till the `e1` becomes a determined logic variable.

Below is a simple example that replicates the Oz program in §1.2. The first `async` function needs both `f` and `g` to be determined. Hence, it will suspend until both logic variables have been provided by the second `async` function. This occurs after two seconds.

```
const f = new LogicVariable();
const g = new LogicVariable();

(async function() {
    console.log((await f) + (await g)); // eventually prints 15
})();

(async function() {
   await delay(2); // suspend for two seconds
   f.unify(new LogicVariable(5));
   g.unify(new LogicVariable(10));
})();
```

[4]The terms 'reference' and 'transient' has been borrowed from Mehl(1999), who uses them with the same meaning

3.2 Threads' abstraction

There are two capabilities that we expect our threads' abstraction to support:

1. Users should be able to launch an arbitrary number of threads
2. Users should be able to adjust the priorities given to various threads, in order to influence their scheduling

Below is a simple code snippet that demonstrates our vision for the threads' abstraction:

```
const matrix = initialiseMatrix();
thread(() => compute(0), // works on row 0
       () => compute(1), // works on row 1
       () => compute(2)); // works on row 2

const compute = rowNumber => { // some calculations on a matrix row }
```

To realise this threading model, I have created a transpiler that transforms the programs written in the JavaScript extension. Every nullary function passed to the `thread` function is converted into an asynchronous function. Suspension points are also inserted into the body of every nullary function. These suspension points trigger timed delays if the thread's time slice has expired, allowing the JavaScript runtime to automatically switch to one of the other nullary functions passed to `thread`.

Since each thread's nullary function becomes an asynchronous function in the generated code, the `thread` function can be implemented with a simple loop that invokes every `async` function. This is shown below.

```
function thread() {
  for (let func of arguments) {
    func({time: performance.now()});
  }
}
```

There is a small amount of book-keeping required, because every thread needs to track the length of time since its resumption. This explains why the `thread` function invokes each `func` with an object containing a time measurement.

4 Evaluation of current design

My current implementation consists of: (1) a JavaScript library that implements logic variables as promise-like objects (shown in §3.1), and (2) a transpiler that implements threads as `async` functions (shown in §3.2) Although it is not yet possible to use these two tools together, this current work does satisfy a subset of the project's goals. The list below shows the current status of the project goals.

1. The logic variables' library and the transpiled code can run on web browsers.
2. Programmers can create, unify and use logic variables in `async functions`.
3. Further work is needed to support pattern matching on logic variables. The idea has been outlined in §2.3.

4. Programmers can launch threads in the JavaScript extension. Further work is needed to support thread management (setting priorities, waiting for threads, terminating threads).

§4.1 discusses the strengths of the current implementation. Over the next semester, I will address the limitations of the current implementation, which are mentioned in §4.2

4.1 Strengths of current design

Logic variables' library

The API provided by the logic variables' library is quite user-friendly. Specifically, the dataflow semantics of the logic variables can be used easily with the `await` keyword.

Transpiler

Because the transpiler implements threads as `async` functions, each thread's state is managed implicitly by the JavaScript runtime. No additional effort is needed, unlike the Doppio runtime where each thread's stack is managed explicitly.

4.2 Limitations of current design

This section describes limitations of the current design, which I will address over the next leg of this project.

Transpiler

The transpiler currently emits correct code only for a subset of JavaScript. Specifically, it does not yet support asynchronous functions and object instantiations in the source language. Implementing this support will be my most immediate task.

In addition, the transpiler cannot add suspension points into third-party JavaScript code. It is hence not possible for threads to suspend when they invoke a long-running third party JavaScript function.

Lastly, the transpiler needs to be further extended to support thread management. My current idea is that each thread will have a priority which will be considered at each suspension point.

5 Areas that need further investigation

This section outlines areas that deserve further investigation over the next leg of the project.

Performance benchmarking

Not much work has been done yet, to evaluate the performance of the current implementation. This will be a priority once the transpiler implementation is complete. Specifically, I need to analyse whether the current implementation of threads can support tail recursion in the JavaScript extension.

Doppio

As discussed earlier, Doppio is a project that has also created a threads' abstraction in browser-based JavaScript. It would be helpful to benchmark Doppio's approach against this project's implementation.

Assessment of web workers

Given that web workers are a key part of the modern JavaScript landscape, they should be assessed more thoroughly as a potential means of implementing the threads' abstraction.

Implementing threads with asynchronous generators

Instead of implementing threads as asynchronous functions, we can implement them as asynchronous generators. Each generator would yield control to an explicit scheduler. This approach deserves investigation and benchmarking, because thread management may be easier to implement with an explicit scheduler. Vilk & Emery (2014) and Beazley (2009) have established that generators can be used to implement threads.

6 Acknowledgements

I wish to thank my supervisor, Professor Martin Henz, for challenging me to strive towards greater goals. I am also grateful to my fellow student, Thomas Tan, for his insightful ideas on the implementation of threads in this language extension.

7 References

1. Beazley, D. (2009, March 25). A curious case on coroutines and concurrency. Retrieved from http://www.dabeaz.com/coroutines/

2. Costa, L. F. (2017, April 30). JavaScript: From Workers to Shared Memory. Retrieved November 1, 2020, from https://lucasfcosta.com/2017/04/30/JavaScript-From-Workers-to-Shared-Memory.html

3. Mehl, M. (1999). The Oz Virtual Machine: Records, Transients and Deep Guards (Doctoral dissertation). Saarland University. Retrieved August 3, 2020, from https://d-nb.info/972333401/34

4. Scheidhauer, R. (1998). Design, Implementierung und Evaluierung einer virtuellen Maschine fr Oz (Doctoral dissertation). Saarland University.
Retrieved from http://mozart2.org/publications/abstracts/scheidhauer-thesis.html

5. Simpson, K. (2015). You Don't Know JS: Async & Performance (1st ed., Vol. 1). O'Reilly.

6. Van Roy, P., & Haridi, S. (2004). Concepts, Techniques and Models of Computer Programming. Cambridge, Massachusetts: MIT Press.

7. Vilk, J., & Berger, E. D. (2014). Doppio: Breaking the Browser Language Barrier (Vol. 35, pp. 508-518). Programming Language Design and Implementation.
doi:https://doi.org/10.1145/2594291.2594293

8. Async / await. (2020, October 29). Retrieved September 10, 2020, from https://javascript.info/async-await

I referred to the open source *z* pattern matching library in §2.3. It is available at: https://github.com/z-pattern-matching/z