

Bachelor Thesis

# Transparent Observables

by Simon Siefke

Supervised by:  
Prof. Dr. Markus Müller-Olm

Münster, June 2019

# Abstract

Its 2019 and creating user interfaces is still hard. This paper will explore reactive programming for web application. The goal is to make building user interfaces easier to build, easier to understand and easier to maintain. Even though this paper focuses on web technologies, the ideas can be useful for every kind of user interface programming.

The web is an especially interesting place because it is open and accessible to everyone on almost all devices. The growing number of features and connections between devices also provides challenges for managing application state.

We present a prototype for a programming language based on JavaScript in which every variable is reactive, just like in Excel spreadsheets.

# Table of Contents

1	Introduction	4
1.1	Motivation	4
1.2	Outline	6
2	Fundamentals	7
2.1	Reactive Programming	7
2.1.1	Observer Pattern	7
2.1.2	Invalidate Pattern	8
2.2	Functional Reactive Programming	9
2.3	Data binding	11
3	Creating a reactive programming language	16
3.1	Syntax and Semantics	16
3.1.1	Assignments	16
3.1.2	Imports and Exports	18
3.1.3	Functions	19
3.1.4	If-Else	19
3.2	Implementation	20
3.2.1	Compiler	20
3.2.2	Library	21
3.2.3	Command Line Interface	23
3.3	Evaluation	24
3.3.1	Possible Optimizations	25
3.3.1.1	Fine-grained reactivity	25
3.3.1.2	Propagating changes effectively	26
3.3.2	Glitch Avoidance	27
4	Conclusion	29
5	List of References	32
6	List of Figures	34
7	List of Abbreviations	35

# 1 Introduction

## 1.1 Motivation

Web applications have a very broad use: One use case are websites. There are approximately 1.5 billion websites [0].

Another extended use case of websites are progressive web apps. Progressive web apps are websites on FIRE [1]:

- Fast – They don't have as much code as normal websites and have advanced caching.
- Integrated – They can be accessed from mobile devices like a native app, with an icon on the home screen and without a browser surrounding the website.
- Reliable – Once loaded, they work offline & with flaky internet connections (via caching).
- Engaging – They can send push notifications like native apps.

Progressive web apps make websites more accessible for users on every device, but especially for people with low-end devices and slow internet connections. Progressive web apps can also be installed on laptops and pc's, which makes them easier to access for laptop and PC users.

Beyond just websites and PWA's, web technologies are also popular for more advanced desktop applications, for example Visual Studio Code by Microsoft [2], Atom by Github [3] both use HTML, CSS and JavaScript (or variants) on top of Electron [4] which is a framework for developing desktop applications that have access to the file system, can execute commands, etc.

More and more application logic is shifted onto frontend javascript applications. For example squoosh.app is an image optimization application developed by GoogleChromeLabs. Efficient compression algorithms written in C++ have been transpiled into webassembly and are executed by the client's browser.

Reactive Programming and Functional Reactive Programming have gained a lot of traction in recent years.

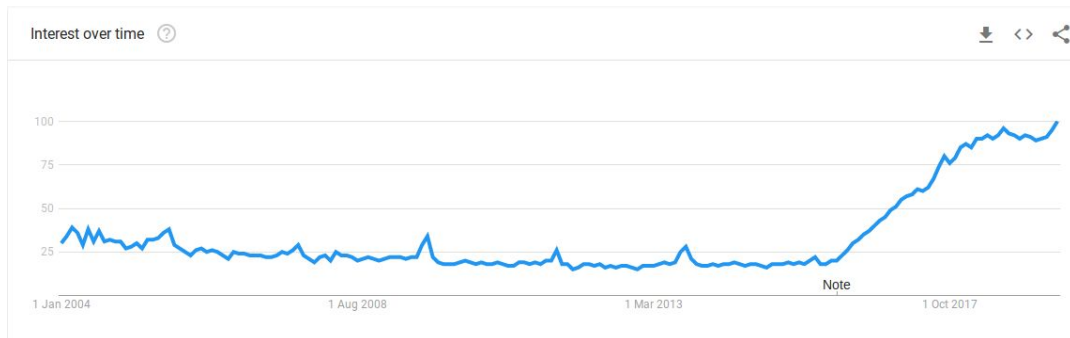


Figure 1.1: Search Interest for Functional Reactive Programming

It has been used for programming Robotics [5], Games [6] and Computer Vision [7].

Compared to programming using the Object Oriented Observer Pattern, Reactive Programming has been shown to be generally easier to comprehend. It was also shown that the entrance barrier for Reactive Programming is lower [8].

Reactive Programming is used in some form or another in many JavaScript frameworks, such as Angular, Vue, React or Svelte. Usually there is at least some amount of boilerplate code involved for using the reactivity API that the framework or the library exposes.

The concept of transparent observables is simple: Using reactive programming in a way that makes working with observables easy by creating observable variables without explicitly registering or unregistering observers for the variable.

Creating a language that deeply Integrates reactivity will be useful for the following reasons:

- Almost every user interface is reactive and needs code that handles reactivity. Therefore observables are crucial for every dynamic user interface
- When reactivity and updating variables is handled automatically, it leads to less error-prone and prevents memory leaks. This is similar to how higher order programming languages don't need to manage memory manually, because it is handled automatically
- Reactivity will be easier to learn and use because it is close to natural language

## 1.2 Outline

The fundamentals section will cover everything that is important for understanding “Transparent Observables”. It will explain what Reactive Programming is, what Functional Reactive Programming is and what Data Binding is. It will also show how these concepts are currently used inside web applications, JavaScript frameworks and JavaScript libraries.

It will introduce the Observable/Observer design pattern and the Invalidate pattern.

Then it will explain how reactivity is used for creating web applications in state of the art frameworks such as Angular, React, Vue and Svelte. This includes Reactive Programming as well as Functional Reactive Programming, 1-way-binding and 2-way-binding.

After this overview of state of the art approaches to reactivity, we will talk about a prototype for a programming language that we created, in which every variable is reactive. We will first cover the syntax before going into the implementation details.

Finally we will evaluate the new programming language and talk about how it compares to current approaches.

## 2 Fundamentals

### 2.1 Reactive Programming

#### 2.1.1 Observer Pattern

The Observer Pattern is the classical approach for using reactivity. It was specified in the book Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [9].

It is used to keep information between components consistent by having one object that holds the state, the subject (also known as observable), and other components register themselves for state changes, the observers. When the state changes, the observable notifies the observers, which can update accordingly. The sequence diagram visualizes the flow of the updating mechanism:

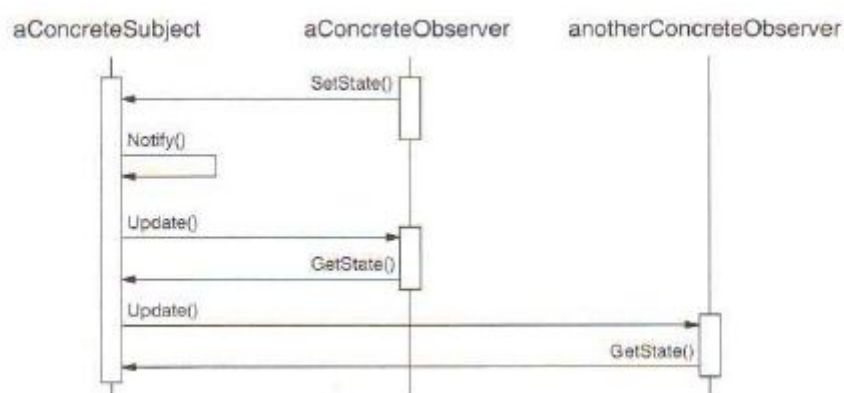


Figure 2.1.1 Sequence Diagram of the Observer Pattern

An implementation of the Observer Pattern might look like this (instead of classes it uses functions, but the idea is the same):

```
function observable(value) {
  const observers = new Set()
  return {
    get() {
      return value
    },
    set(newValue) {
      value = newValue
      observers.forEach(fn => fn())
    },
    subscribe(fn) {
      observers.add(fn)
    },
  }
}

function combineLatest(a, b, callback) {
  const result = observable(callback(a.get(), b.get()))
  a.subscribe(() => result.set(callback(a.get(), b.get())))
  b.subscribe(() => result.set(callback(a.get(), b.get())))
  return result
}

const a = observable(1)
const b = observable(2)
const sum = combineLatest(a, b, (a, b) => a + b) // initial sum
value is set to 3
a.set(23) // sum value is updated to 25
b.set(b.get() + 1) // sum value is updated to 26
```

For the purposes of this paper we will define an observable as a reactive value. The observable can have other reactive values, called observers, that depend on the value of one or more observables.

### 2.1.2 Invalidate Pattern

The invalidate pattern is similar to the observer pattern in the sense that it is used to keep state in consistent.

```
let a = 1
```



```
let b = 2
let sum
function invalidate(){
  sum = a + b
}
invalidate() // sum is 3
a = 23
invalidate() // sum is 25
b++
invalidate() // sum is 26
```

When the invalidate pattern is used, the developer must call the `invalidate` function to update the `sum` value. Using the observer pattern, this is done automatically. Another difference is that the observer pattern wraps values inside objects which makes accessing the values more complicated because of the get/set boilerplate code. The invalidate pattern also allows for multiple changes of a variable before dependent expressions are recalculated.

## 2.2 Functional Reactive Programming

Functional reactive programming was first proposed in 1997 by Conal Elliot and Paul Hudak as a solution for a more declarative way for creating animations [10].

The two central abstract types in FRP are Behaviours and Events.

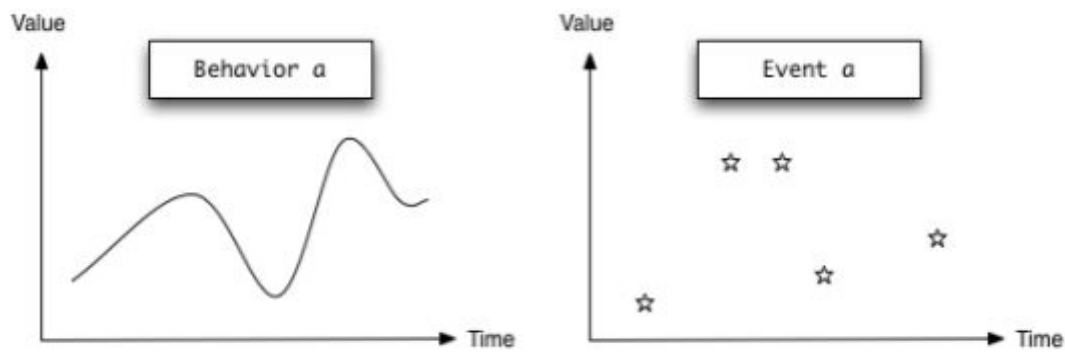


Figure 2.2.1: Behaviours vs. Events

Conal Elliot and Paul Hudak defined them as [10]:

*Behaviours represent values that change in time – for example, mouse X coordinate varies in time, but it always has some value.*

*Events represent discrete events in the system – they happen every now and then and can trigger some change, but do not always have a value. For example, mouse click can happen, but you cannot ask "what's the current value of click".*

Imagine a simple application that just reverses a given string.

- There is an input field, where the user can type the word that should be reversed
- The reversed word is displayed below the input field

Hello World  
dlroW olleH

Figure 2.2.2: Reverse Text Sample – [Edit on CodeSandbox](#)

Implementing this with Flapjax [11], an FRP language for the web, could look like this:

```
<!DOCTYPE html>
<html>
  <body>
    <input type="text" id="input" />
    <br>
    <div id="output"></div>
    <script
src="https://www.flapjax-lang.org/fx/flapjax.js"></script>
    <script>
      const inputB = extractValueB('input', 'value')
      function reverse(string) {
        return (string || '')
          .split('')
          .reverse()
          .join('')
      }
      const reverseStringB = inputB.liftB(reverse)
      insertDomB(reverseStringB, 'output')
    </script>
  </body>
</html>
```

`extractValueB` is the function that creates the behaviour stream of input events. Using `liftB` the stream is then transformed into another stream that emits the reversed input value. Every time the `reverseTextStream` emits a value, its subscriber callback function is invoked, which updates the output to the reversed value of the input.

This is a very declarative way of defining an application. Also notice that each behaviour stream is only defined once, at one place and not mutated. Event streams and Behaviour Streams can be composed to create new streams, just like functions can be composed in Functional Programming.

## 2.3 Data binding

Forms inputs with custom validation or autocomplete are very common nowadays. To create a form with custom validation or autocomplete, it is

necessary to have variables in the application code that correspond to values form input fields.

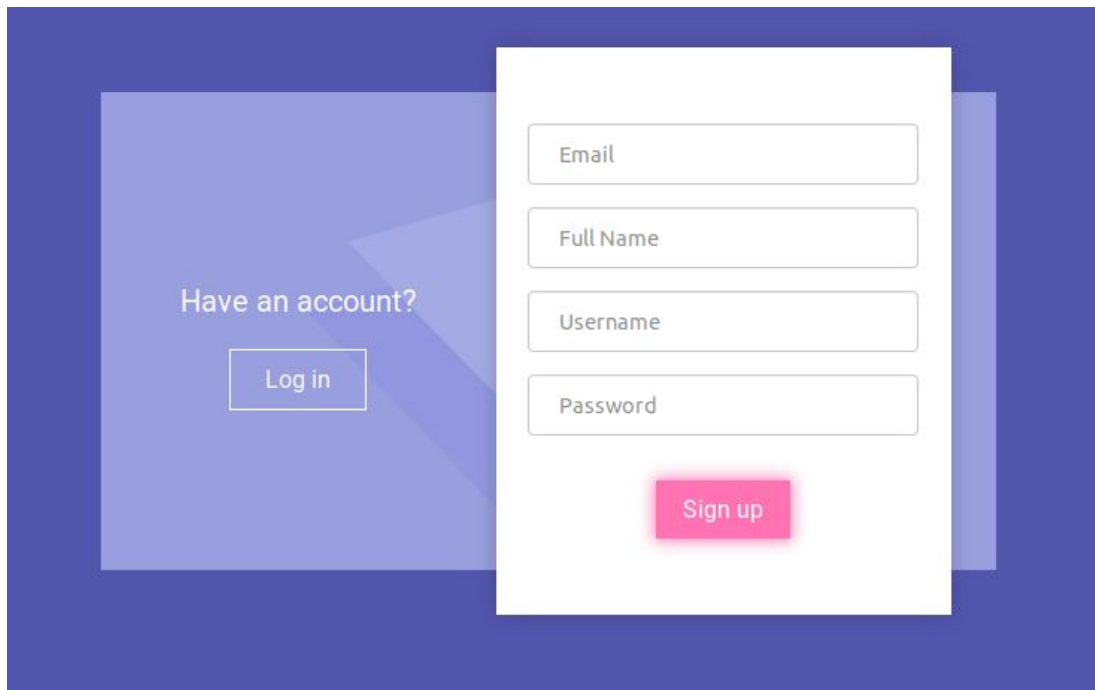
The image shows a user interface for a sign-up form. It features a dark blue background. On the left, there's a semi-transparent blue rectangle containing the text "Have an account?" and a "Log in" button. Overlaid on the right is a white rectangular form. This form contains four input fields labeled "Email", "Full Name", "Username", and "Password". Below these fields is a prominent pink "Sign up" button.

Figure 2.3: Sign up Form - [Edit on Codepen](#)

For this sign up form, we might want to have

- an email variable, whose value corresponds to the Email input field
- a fullName variable, whose value corresponds to the Full Name input field
- a userName variable, whose value corresponds to the Username input field
- a password variable, whose value corresponds to the Password input field

The idea of 2-way-binding is just this: Keeping the value of a variable in sync with the value of another thing, e.g. the value of an input field.

In VueJS for example, 2-way-binding can be achieved using the v-model directive:

```

<template>
<div>
  <input v-model="email" placeholder="Email">
  <input v-model="fullName" placeholder="Full Name">
  <input v-model="userName" placeholder="Username">
  <input v-model="password" placeholder="Password">
</div>
</template>

<script>
export default {
  data() {
    return {
      email: "",
      fullName: "",
      userName: "",
      password: ""
    }
  }
}
</script>

```

1-way-binding is similar to 2-way binding, it is often used for templating part in JavaScript frameworks, for example in Vue:

```

<template>
<span>the time is {{time}}</span>
</template>

<script>
export default {
  data() {
    return {
      time: new Date()
    }
  },
  mounted() {
    setInterval(() => {
      this.time = new Date()
    }, 1000)
  }
}
</script>

```

[Edit on CodeSandbox](#)

Vue handles updating the template part and ensures that it always displays the value of the time variable, the `{{time}}` inside the template part is bound to the javascript time variable.

Data binding is in close relation with the observer pattern/observables. Both principles are used complementary in frameworks like Angular, Vue or Svelte. Data binding is almost always used for inputs and the templating part, which is part of the component's state. Observables are used for non-input event related application state.

In the VueJS example above, the time variable inside the data section is a reactive value/observable. Vue just uses a less verbose syntax than this:

```
let time = observable(new Date())
setInterval(() => {
  time.set(new Date())
}, 1000)
```

Because data binding is a simple yet powerful concept, using data binding for variables might be the norm in 2051, at least according to Paul Stovell [12]:

*In the year 2051, reactive programming is the norm. Language creators discovered the **destiny operator** decades ago, and the old ways were quickly forgotten. For example, in P#, we can write:*


```
var a = 10;
```

```
var b <= a + 1;
```

```
a = 20;
```

```
Assert.AreEqual(21, b);
```

*As you can see, the statement establishes **b** and **a** as having intertwined destinies, which are unbroken and forever. They are **bound**. The relationship between them isn't implicit, an idea that only exists in the*



*mind of the programmers; it's explicit, a part of the language, and it exists for all time.*

## 3 Creating a reactive programming language

The reactive programming language that we created is named “Relevant UI” or “UI” for short. Unfortunately, everything that has “react” in its name was already taken.

Besides finding a name, the second most important task is specifying what the language does, what it doesn’t and implementing it. We will cover that in the following sections.

### 3.1 Syntax and Semantics

#### 3.1.1 Assignments

Assignments are reactive. They don't require the keyword ``var``, ``let`` or ``const`` in front of them (like they would in JavaScript in strict mode). The reason for this is that it would be unclear in certain cases whether a variable is a constant or not, for example:

```
x = 0
y = x + 1
x = 1
```

``y`` could be seen as constant because it is not reassigned anywhere in the source code but ``y`` could also be not seen as a constant because it would be reassigned when `x` is reassigned.

Every assignment has a variable Identifier on the left side, an equal sign in the middle and an expression on the right side (an expression can be basically anything: a number, an arithmetic expression, a function invocation, etc.).

There are 3 types of assignments:



**Type 1:** When the expression on the right hand side is constant or contains the variable on the left hand side, it is normal assignment. This means that it is only executed once.

**Example 1:** The variable `x` is assigned a value and then it is assigned another value

```
x = 1
x = x + 1 // updates the value of x to 2
```

**Type 2:** When the expression on the right hand side is a variable, it is a 2-way-binding assignment. This means that the variable on the left and the variable on the right will always have the same value. Once a variable is 2-way-bound it cannot be 1-way-bound to another expression (type 3 assignment), it can only be reassigned through type 1 assignments.

**Example 2:** The variable `y` is 2-way-bound to the variable `x`

```
x = 1
y = x
y = y + 1 // updates x and y
x = x + 1 // updates x and y
```

**Type 3:** When the expression on the right hand side doesn't contain the variable on the left hand side and it isn't a Type 2 assignment, the variable on the left hand side is 1-way-bound to the expression on the right hand side. This means that whenever a variable used on the right hand side changes then the variable on the left hand side is updated again to the value of the expression.

**Example 3:** The variable `y` is 1-way-bound to the expression `x+1`

```
x = 0
y = x + 1
x = 12 // updates the value of x and y
```

Also a variable can only be bound to one expression. The problem is that otherwise the reactivity of the variables would either be compromised or it would lead to unexpected results, for example:

```
// bad code
x = 0
y = x + 1
y = 1
x = 12
```

One possible way to evaluate this code is that in line 3, `y` would be assigned the value 1 and that it is now bound to the value 1. The assignment in line 4 would not change the value of `y`. This means that the reactivity of line 2 would be lost.

The other possible way to evaluate the code is that in line 4, `y` would still be bound by the assignment in line 2 and that it would reactively update to 13. This would be confusing because in line 3 it was assigned the value 1 and now it doesn't have the value 1 anymore.

Since neither evaluation would make sense, this code is just invalid.

### 3.1.2 Imports and Exports

In JavaScript, things can be imported and exported from files to share code between modules and encapsulate logic. Many programming languages do this. In UI, imports and exports are also available and they are reactive. A syntactic difference to JavaScript is that the keywords `let` and `const` are not needed.

Here is an example:

```
// count.ui
export count = 0
setInterval(() => {
  count++
}, 1000);
```

```
// main.ui
import { count } from './count.ui'
document.body.innerHTML = count // updates every second
```

### 3.1.3 Functions

Functions can not only return a value, they can return a value that changes over time:

```
// main.ui
function createCount(){
  count = 0
  setInterval(() => {
    count++
  }, 1000)
  return count
}

document.body.innerHTML = createCount() // updates every second
```

Even though this can be seen as just being syntactic sugar for using observables, it can also be seen as a different way of thinking about functions and what functions can do.

### 3.1.4 If-Else

Conditionals can be expressed the same way as in JavaScript. The condition must be reevaluated whenever a variable inside the condition changes.

```
x = 1
if (x > 2) {
  document.body.innerHTML = `x is big, because its value is ${x}`
} else {
  document.body.innerHTML = `x is small, because its value is ${x}`
}
setInterval(() => x++, 1000)
```

In this example it would first display that `x` is small and after two seconds it would display that `x` is big.

## 3.2 Implementation

This part covers the implementation of Reactive UI. Not all features of the specification could be implemented, however for a prototype it works good enough. Specifically 2-way-binding was only partially implemented, reactive imports and exports were not implemented and reactive assignments only work for primitive values (strings, numbers, etc.).

The source code for the implementation is [available on Github](#) [13].

Source code written in UI will be transformed into JavaScript which can then be executed in a browser or anything that can execute JavaScript. The goal of the generated code is to make assignments reactive. When a variable changes, dependent variables must update. To achieve this we created a library, a compiler and a command line interface:

- The library includes the updating mechanism/function.
- The compiler inserts extra code that imports and calls the update mechanism of the library.
- The command line interface enables compilation via the command line.

The basic idea for the updating mechanism and the compiler stems from SvelteJS [14]. The idea was extended to make every variable reactive and to 2-way-bind variables with a simple assignment.

### 3.2.1 Compiler

The current version of the compiler uses regular expressions to transform UI code to JavaScript.

For example this code:

```
x = 0
```

```
y = x + 1
x = 22
```

is transformed into

```
import { updates, invalidate, dirty } from '@relevant/ui'

let x = 0
let y = x + 1
x = 22; invalidate('x');

function update(){
  if(dirty.has('x')){y = x + 1;invalidate('y')}
}
updates.push(update)
```

At the top are 3 imports from the Relevant UI library. Then comes the original source code with the difference that after a change of a variable, the `invalidate` function is called with the name of the variable as an argument. At the bottom of the file is an `update` function, which is added to the `updates` array imported from the ui library. The `update` function reassigns `y` when `x` was changed. Then it marks `y` as changed by calling the `invalidate` function.

### 3.2.2 Library

To further understand how it works, we will inspect the code of the ui library.

```
/**
 * Set of variables that are currently dirty / have recently
 * changed.
 */
export const dirty = new Set()

/**
 * The scheduled update if there is one.
 */
let scheduledUpdate: NodeJS.Timeout | undefined

/**
 * An array of update functions, each file has an update function.
 */
```

```

export const updates: Function[] = []

/**
 * Updates everything.
 */
function updateAll(): void {
  for (const update of updates) {
    update()
  }
  dirty.clear()
  scheduledUpdate = undefined
}

/**
 * Marks a variable as dirty and triggers an update so that other
 * variables that depend on this variable update.
 *
 * @param variableName - The name of the variable that changed.
 */
export function invalidate(variableName: string): void {
  dirty.add(variableName)
  if (!scheduledUpdate) {
    scheduledUpdate = setTimeout(updateAll, 0)
  }
}

```

The main thing the library code does is calling `updateAll` when a variable is changed/invalidated. The updating logic for the individual modules is the `update` function of each module. The `updateAll` function just calls the `update` function of every module. One interesting thing to say about the library code is the line `scheduledUpdate = setTimeout(updateAll, 0)`. This would be similar to `updateAll()`, but the difference is that `scheduledUpdate = setTimeout(updateAll, 0)` is run in the next iteration of the JavaScript event loop. For example:

```

setTimeout(() => {
  console.log('hello')
}, 0)
console.log('world')

```

This would first log `world` and then `hello`.

What this means for the relevant library is that the `updateAll` function is always called on the next iteration of the event loop, after every `invalidate` function call of the current iteration of the event loop is done.

The code for the `y = x + 1` example at the beginning of this chapter would be executed in the following order:

1. `x` is assigned the value `0`
2. `y` is assigned the value `x + 1`, which is `1`
3. `x` is assigned the value `22`
4. `invalidate` is called
5. A timeout is scheduled for the invocation of the `updateAll` function
6. The current iteration of the event loops ends and the next iteration begins
7. The `updateAll` function is called, which calls the `update` function
8. `y` is assigned the value `x + 1`, which is `23`

### 3.2.3 Command Line Interface

A command line interface was created for compiling relevant ui files. The Cli is also written in typescript, and uses the Commander npm package [15] as well as the `@relevant/compiler` and the `@relevant/ui` package.

To use the cli, it must first be installed via npm:

```
npm i -g @relevant/cli
```

Then relevant ui files can be compiled via the command line.

```
relevant index.ui
```

Currently it the CLI can only take one filename or path as an argument and which it compiles into JavaScript.

### 3.3 Evaluation

The goal of this paper was to make building user interfaces easier to build, easier to understand and easier to maintain by creating a programming language in which every variable is reactive.

With Relevant UI, a proof of concept has been created for such a language. The main advantage of Relevant UI compared to similar approaches like Angular, React, Vue or Svelte is that there is zero boilerplate for creating reactive variables.

As already described in the implementation section, there are still a lot of parts that would need to be implemented in order to fulfill the specification requirements.

Another disadvantage is that there are no tools related to Relevant UI. This means no syntax highlighting, no intellisense, no formatting, no static analysis / error checking, no rename variables with F2, no automatic imports, no type system (TypeScript language features can be used in JavaScript for type checking [16]).

It remains an open question whether or not it is a good idea to make every variable reactive, the main concern is whether or not this makes algorithms more confusing, for example:

```
// bubblesort.ui
function bubblesort(array) {
  for (i = 0; i < array.length; i++) {
    for (j = 0; j < array.length; j++) {
      if (array[i] > array[j]) {
        tmp = array[i]
        array[i] = array[j]
        array[j] = tmp
      }
    }
  }
}
```

The assignment `array[i] = array[j]` would also change the `tmp` variable which means that `array[j] = tmp` does not assign `array[i]`.



### 3.3.1 Possible Optimizations

This section covers how a reactive language can be optimized. Currently, none of the following optimizations are implemented in Relevant UI.

#### 3.3.1.1 Fine-grained reactivity

MeteorJS is a JavaScript Framework with a reactive system called “Tracker”, whose purpose is to manage application state in a reactive way, for example rerendering the user template when the user record in the database changes.

Imagine the user is just an object with a name and an age, and the template only displays the name of the user. With some a simple reactive implementation the template would be rerendered whenever something in user object changes, even though it would just need to rerender when the user name changes.

Here is an example from the Tracker documentation [17]:

```
var data = new ReactiveDict;

// VERSION 1 (INEFFICIENT)
Tracker.autorun(function () {
  if (data.get("favoriteFood") === "pizza")
    console.log("Inefficient code says: Time to get some
pizza!");
  else
    console.log("Inefficient code say: No pizza for you!");
});

// VERSION 2 (MORE EFFICIENT)
Tracker.autorun(function () {
  if (data.equals("favoriteFood", "pizza")) // CHANGED LINE
    console.log("Efficient code says: Time to get some
pizza!");
  else
    console.log("Efficient code says: No pizza for you!");
});
```

The first version ("Inefficient") prints a message whenever "favoriteFood" changes. The second ("More efficient") version is finer-grained: it prints a message only when "favoriteFood" changes to or from "pizza".

### 3.3.1.2 Propagating changes effectively

Optimizing the propagation of changes in a reactive system can lead to performance benefits when a lot of data is reactive. For simplicity, we will consider a small example:

```
words = ['lorem', 'ipsum']  
upperCaseWords = words.map(word => word.toUpperCase()) //  
['LOREM', 'IPSUM']
```

When another word is added to the `words` array, the `upperCaseWords` array would need to be recomputed. The simple approach would be reevaluating:

```
upperCaseWords = words.map(word => word.toUpperCase())
```

whenever the `words` array changes, which takes  $O(n)$  time for every insertion.

There is another approach, that leads to  $O(1)$  time for inserting a new word into the `words` array and updating the `upperCaseWords` array:

When `upperCaseWords` is first assigned, just evaluate the assignment. This takes  $O(n)$  time and cannot be optimized. When a word is added to the `words` array, e.g:

```
words = [...words, 'dolor'] // ['lorem', 'ipsum', 'dolor']
```

assign the `upperCaseWords` array in the same way, applying the transformation only to the added element:

```
upperCaseWords = [...words, (word => word.toUpperCase())('dolor')]
```

```
// ['LOREM', 'IPSUM', 'DOLOR']
```

This code might look weird to non-JavaScript programmers because of the immediately invoked function expression. So here is the same code separated into two statements:

```
transformation = word => word.toUpperCase()  
upperCaseWords = [...words, transformation('dolor')] // ['LOREM',  
'IPSUM', 'DOLOR']
```

Generally speaking: For a reactive array:

- mapping  $k$  insertions can be done in  $O(k)$
- mapping  $k$  deletions can be done in  $O(k)$

assuming that there only are a constant number of dependent variables and that the time complexity for the transformation function is also constant. The transformation function should also be side-effect free.

Note that there can be done similar optimizations for

- sorting: inserting into a sorted array instead of inserting and resorting
- searching: inserting into a binary search tree instead of inserting and traversing the whole array
- filtering: apply filter-function to inserted element instead of reiterating the whole array

### 3.3.2 Glitch Avoidance

One important thing when implementing a reactive language is to avoid glitches. Wikipedia explains it pretty well [18]:

When propagating changes, it is possible to pick propagation orders such that the value of an expression is not a natural consequence of the source program. We can illustrate this easily with an example. Suppose `seconds` is a reactive value that changes every second to represent the current time (in seconds). Consider this expression:

$$t = \text{seconds} + 1$$
$$g = (t > \text{seconds})$$

Because `t` should always be greater than `seconds`, this expression should always evaluate to a true value. Unfortunately, this can depend on the order of evaluation. When `seconds` changes, two expressions have to update: `seconds + 1` and the conditional. If the first evaluates before the second, then this invariant will hold. If, however, the conditional updates first, using the old value of `t` and the new value of `seconds`, then the expression will evaluate to a false value. This is called a glitch.

In Relevant UI, glitches are avoided by using the Invalidate pattern. When a variable references another variable, it must be below that variable in the source code, because it cannot reference a variable that is not yet defined. When that is the case, the updating of part of that variable is after the updating part of the dependent variable. Because of that, it is updated later.

For the `t = seconds + 1` example this means that `t = seconds + 1` is always evaluated before `g = (t > seconds)` and the invariant holds.

## 4 Conclusion

With Svelte being the first mainstream web framework creating a language that uses transparent observables to create user interfaces for the web, it is likely that there will come more programming languages in the future that have the power of transparent observables built in. And they will likely come in many different flavours, the same way programming languages nowadays do. There are the simple ones with an easy to learn and readable Syntax like Python or CoffeeScript for the web. There are the Functional ones like Haskell or Elm for the web. With Type systems or no type systems or optional type systems.

Also as transparent observables are very close to natural language, there might be a possibility to derive the code for the application directly from a natural language specification:

"Make a button":

```
<button></button>
```

"Create a variable count and when the button is clicked, increment count by 1"

```
<script>
  let count
</script>
<button on:click="count++"></button>
```

"Oh and count initially has the value 0"

```
<script>
  let count = 0
</script>
<button on:click="count++"></button>
```

"Oh and display the count inside the button"

```
<script>
  let count = 0
</script>
<button on:click="count++">{count}</button>
```

Even though this probably won't happen any time soon, it is interesting to think about how easy programming could become. And if it would work, it would scale very well because of the way components can be structured: Ideally there are just a lot of small, independent components for a large application. The difficult part is orchestrating these components and managing application state between them.

It is also probable that there will be more API's or libraries written in or at least compatible with a language that has transparent observables. For example it would be useful if there was a reactive version of the localStorage API whose stored values are bound to JavaScript variables, e.g.

```
<button on:click={() =>count++}>
  {count}
</button>

<script>
  count = localStorage.count = localStorage.count || 0
</script>
```

In this example the variable count would be 2-way bound to the localStorage, which means when `count` changes, the value of `count` is persisted into localStorage and when localStorage changes, the value of `count` would be updated as well. This also means that the count is synchronized between all tabs and restored when a page is closed and reopened.

Currently one way to do that (in svelte) would be:

```
<button on:click={()=>count++}>{count}</button>

<script>
  let count = JSON.parse(localStorage.getItem("count") || "0");
  window.addEventListener("storage", event => {
    count = JSON.parse(event.newValue);
  });
  $: {
    localStorage.setItem("count", JSON.stringify(count));
  }
</script>
```

[Edit on CodeSandbox](#)

There is a lot of potential for cleaner code not only for the localStorage API, but also for

- Databases in general
- WebRTC, Websockets and live data, online multiplayer games

## 5 List of References

[0] Internet Live Stats, Total number of Websites

URL: <https://www.internetlivestats.com/total-number-of-websites/>

Accessed: 2019-05-27

[1] J. Posnik. Beyond SPAs: alternative architectures for your PWA

URL: <https://developers.google.com/web/updates/2018/05/beyond-spa>

Accessed: 2019-05-05

[2] Microsoft, VSCode Text Editor

URL: <https://github.com/microsoft/vscode>

Accessed: 2019-05-27

[3] Github, Atom Text Editor

URL: <https://github.com/atom/atom>

Accessed: 2019-05-27

[4] Github, Electron Framework

URL: <https://github.com/electron/electron>

Accessed: 2019-05-27

[5] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming.

URL: [www.cs.yale.edu/homes/hudak/CS429F04/AFPLectureNotes.pdf](http://www.cs.yale.edu/homes/hudak/CS429F04/AFPLectureNotes.pdf)

Accessed: 2019-05-27

[6] H. Nilsson, A. Courtney, and J. Peterson. Functional Reactive Programming, Continued.

URL:

<http://haskell.cs.yale.edu/wp-content/uploads/2011/02/workshop-02.pdf>

Accessed: 2019-05-27

[7] J. Peterson, P. Hudak, A. Reid, G. Hager, 2001. Fvision: A declarative language for visual tracking. In Practical Aspects of Declarative Languages: Third International Symposium, PADL 2001 Las Vegas, Nevada, March 11-12, 2001 Proceedings (pp.304-321)

URL: [https://link.springer.com/chapter/10.1007%2F3-540-45241-9\\_21](https://link.springer.com/chapter/10.1007%2F3-540-45241-9_21)

Accessed: 2019-05-15

[8] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini

On the Positive Effect of Reactive Programming on Software

Comprehension: An Empirical Study



URL:

[http://www.guidosalvaneschi.com/attachments/papers/2017\\_On-the-Positive-Effect-of-Reactive-Programming-on-Software-Comprehension-An-Empirical-Study\\_pdf.pdf](http://www.guidosalvaneschi.com/attachments/papers/2017_On-the-Positive-Effect-of-Reactive-Programming-on-Software-Comprehension-An-Empirical-Study_pdf.pdf)

Accessed: 2019-05-27

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 293 ff. ISBN 0-201-63361-2

[10] C. Elliott and P. Hudak. Functional reactive animation. In International Conference on Functional Programming, 1997.

URL: <http://conal.net/papers/icfp97/>

Accessed: 2019-05-05

[11] Flapjax Authors, Flapjax repository

URL: <https://github.com/brownplt/flapjax>

Accessed: 2019-05-15

[12] Paul Stovell, What is Reactive Programming?

URL: <http://paulstovell.com/blog/reactive-programming>

Accessed: 2019-05-27

[13] Source Code for the implementation

URL: <https://github.com/SimonSiefke/transparent-observables>

Accessed: 2019-05-15

[14] R. Harris and Svelte Contributors, Svelte

URL:

<https://github.com/sveltejs/svelte/blob/master/src/compiler/compile/Component.ts>

[15] TJ Holowaychuk and Contributors, Commander.js Library

URL: <https://www.npmjs.com/package/commander>

Accessed: 2019-05-15

[16] Microsoft, Type Checking JavaScript Files

URL:

<https://github.com/Microsoft/TypeScript/wiki/Type-Checking-JavaScript-Files>

Accessed: 2019-05-15

[17] MeteorJS Team, Meteor Tracker

URL:

<https://github.com/meteor/docs/blob/version-NEXT/long-form/tracker-manual.md#reactivedictget-versus-equals>

Accessed: 2019-05-05

[18] Wikipedia, Reactive programming

URL: [https://en.wikipedia.org/wiki/Reactive\\_programming](https://en.wikipedia.org/wiki/Reactive_programming)

Accessed: 2019-05-15

## 6 List of Figures

[Figure 1.1] Search Interest for Functional Reactive Programming, Image by Google Trends

<https://trends.google.com/trends/explore?date=all&q=frp>

[Figure 2.1.1] Sequence Diagram of the Observer Pattern, E. Gamma, R. Helm, R. Johnson, J. Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 295. ISBN 0-201-63361-2.

[Figure 2.2.1] Behaviours vs Events, Images by Heinrich Apfelmus

<https://github.com/HeinrichApfelmus/frp-guides/blob/master/apfelmus/img/frp-behavior.png>

and

<https://github.com/HeinrichApfelmus/frp-guides/blob/master/apfelmus/img/frp-event.png>

[Figure 2.2.2] Reverse Text Example, Screenshot by me

[Figure 2.3] Sign up From, Codepen by Anna Batura

[https://codepen.io/Anna\\_Batura/pen/QEAqyE](https://codepen.io/Anna_Batura/pen/QEAqyE)

## 7 List of Abbreviations

- FRP – Functional Reactive Programming
- DOM – Document Object Model
- WebRTC – Web Real-Time Communication
- HTML – Hypertext Markup Language
- CSS – Cascading Style Sheets