



Projection of Reactive Programming onto Dataflow Engines

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in de Toegepaste Informatica

Alexander Moerman

Promoter: Prof. Dr. Wolfgang De Meuter
Advisor: Mathijs Saey, Florian Myter and Thierry
Renaux

Academic year 2016-2017



Abstract

Declaration of Originality

I hereby declare that this thesis was entirely my own work and that any additional sources of information have been duly cited. I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this thesis has not been submitted for a higher degree to any other University or Institution.

Acknowledgements

Contents

- 1 Introduction**
- 2 Background**
 - 2.1 Reactive Programming 3
 - 2.1.1 Example 4
 - 2.1.2 Advantages 4
 - 2.2 Dataflow Programming 5
 - 2.2.1 Introduction 5
 - 2.2.2 Example 5
- 3 Language**
- 4 Engine**
- 5 Evaluation**
- 6 Future work and limitations**
- 7 Related work**
- 8 Conclusion**
- A Your Appendix**

List of Figures

List of Tables

1

Introduction

2

Background

2.1 Reactive Programming

Reactive Programming is a software development paradigm focused on reactions, i.e. the handling of external events, user interactions, etc. In this paradigm, the application state is derived from the previous state and any events that may occur, for example user interactions or current environmental factors. This deviates from more traditional approaches, where values and state can be written at any point and for any reason. In a reactive program however, the flow of dependencies between values must be recorded once, which can be seen as a (possibly cyclic!) directed graph. When a new value is recorded, it is appended to the end of the graph, where the new node "subscribes" to values from the source nodes. Whenever these source nodes produce new data, they will notify this node, at which point it can recompute what its own value should be. In this way, values ripple through the direct graph of nodes, updating nodes wherever they pass through. Note that values can only ever be appended at the end of the graph, i.e. add more derived values. Once such a "reactive node" - often called an observable or signal in literature - has been defined, it is impossible to modify the source nodes which will feed it data downwards in the graph.

2.1.1 Example

The canonical metaphor for Reactive Programming is spreadsheets, which typically track changes across input cells and automatically recompute values in other cells if the formulas they contain reference the aforementioned input cells. In essence, cells "react" to modifications made in other cells if their formulas depend on them. Imagine a simple program in an imperative programming setting:

```
a = b + c
```

When this statement is executed, it assigns the result of adding *b* and *c* to the variable *a*, effectively mutating *a*. Note that this only happens once. A snapshot is taken of the current value of *b* and *c*, to determine the new value of variable *a*.

In a reactive programming setting, *a* would subscribe to the values of *b* and *c*, essentially asking to be notified whenever the variables *b* or *c* change, at which point the value of variable *a* changes. This process repeats every time the variables *b* or *c* are modified. Note that the value of *a* is undetermined until both *b* and *c* produce a value.

The implementation of this reactive mechanism can be provided by the language itself or by a framework or library.

2.1.2 Advantages

A signal can be described as "values over time", in contrast with a variable which only holds its latest value, revealing no information about the time that value was provided or what changed it. As it turns out, signals can be used to model almost any concept in software development:

- mouse movements as a signal which emits the current position in real time
- click events as a signal which emits event objects
- the results of a database query as a signal which emits only one value
- an infinite sequence as a signal which never stops emitting

Even though the underlying mechanism will still be identical to more traditional approaches (attaching event listeners to DOM events in HTML, opening and connecting to a WebSocket connection, etc.), the fact that all these concepts can be brought together under a single umbrella called "signals" allows for the modeling of higher order operators to map, combine and filter these flows of values in ways that were previously a lot harder.

2.2 Dataflow Programming

2.2.1 Introduction

Dataflow Programming is a paradigm focused on the optimal, parallel execution of functions. In this paradigm, functions are seen as isolated units of code which should be able to execute whenever the necessary parameters have been provided. Contrary to imperative programming, functions are therefore not called directly, but rather whenever all of the parameters are present. The output of that function is then pushed into the "parameter queue" again, ready to be sent to the next function which takes it as its input. Function parameters are typically wrapped in tokens, which carry metadata information about which execution context they belong to, to isolate multiple calls to the same function from one another. In this way, the execution of functions in Dataflow Programming can also be seen as a direct graph of nodes where each node represents a function and each edge represents the output of a function being sent to another function. It is up to the dataflow engine to orchestrate the queue of tokens and functions so that functions are executed correctly and in the correct order. A large difference with Reactive Programming is that Dataflow Programming puts the function call at the center stage, while Reactive Programming puts forward signals as the core concept of its paradigm. In other words, while both systems have the notion of a dependency graph, the nodes in their graphs carry different concepts: function calls and signals respectively.

2.2.2 Example

3

Language

4

Engine

5

Evaluation

6

Future work and limitations

7

Related work

8

Conclusion



Your Appendix

References