Vrije Universiteit Brussel

FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Projection of Reactive Programming onto Dataflow Engines

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Applied Computer Science

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

List of Figures

List of Tables

1
Introduction

2 Background

2.1 Introduction

This research builds on two existing paradigms in software development: reactive programming and the dataflow model. While intimate knowledge about these concepts is not required, a basic understanding of both will be necessary to follow the ideas and implementation of this dissertation. Reactive Programming is situated in the category of higher level software development, serving as an abstraction tool for events and reactions to those events. The dataflow model on the other hand can be considered more 'low level', providing a strategy for the implicitly parallel execution of programs.

2.2 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 is recorded as a (possibly cyclic!) directed graph, making the derivation of the application state very explicit. At the core of reactive programming are the following main concepts:

- The first-class reification of events (making events a first-class citizen)
- The composition of these events through lifted functions
- The automatic tracking of dependencies and re-evaluation by the language runtime

A number of implementations exist for reactive programming, in this thesis we will focus on the interpretation taken in FrTime (Cooper & Krishnamurthi, 2006).

2.2.1 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. These cells are what we call *observables* or *signals* in Reactive Programming. 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, mutating a in the current scope. 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. Of course, this assumes that the variable b and c are provided to the program.

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. See figure 2.1 for the reactive

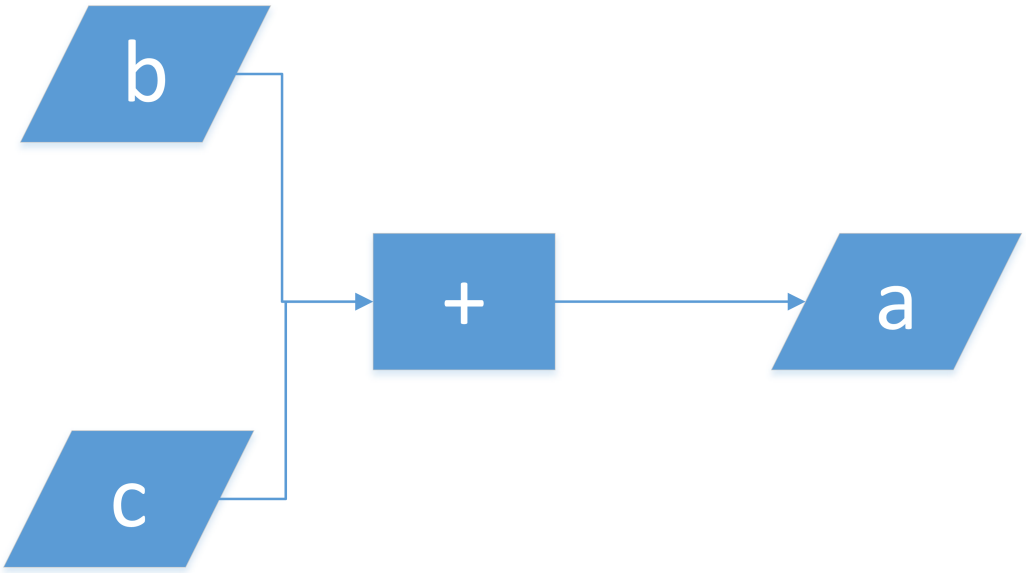

Figure 2.1: Graph of signals

graph. 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.

2.2.2 Advantages of Reactive Programming

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. 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.

2.3 The Dataflow Model

2.3.1 Introduction

The dataflow model is a paradigm focused on the parallel execution of programs. In this paradigm, instructions are seen as isolated units, which should be able to execute whenever the necessary parameters have been provided. Contrary to imperative programming, instructions are not invoked by a program counter, but rather whenever all of the parameters are present.

The execution of instructions in the dataflow model can be seen as a direct graph of nodes where each node represents an instruction and each edge is the output being sent to the next instructions that require the output as arguments. It is up to the dataflow engine to orchestrate the flow of arguments so that instructions are invoked correctly and in the correct order.

Whenever an instruction is invoked, the output is sent through to all connected instructions which depend on it. In Tagged Token Dataflow systems, instruction arguments are wrapped in tokens, which carry meta data about which execution context they belong to in order to isolate multiple calls to the same instruction from one another.

A large difference with Reactive Programming is that Dataflow Programming puts the instruction invocation 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: instructions and signals respectively.

2.3.2 Example

Imagine a simple program in an imperative programming setting:

```
a = b + c
d = a + b
```

This assumes that the variable b and c are provided to the program. In a traditional execution, the variable a would be set to the sum of b and c and the variable d would be set to the sum of a and b. Note that the sequence in which these operations are executed is of vital importance: switching the two statements would result in different values for the variable d!

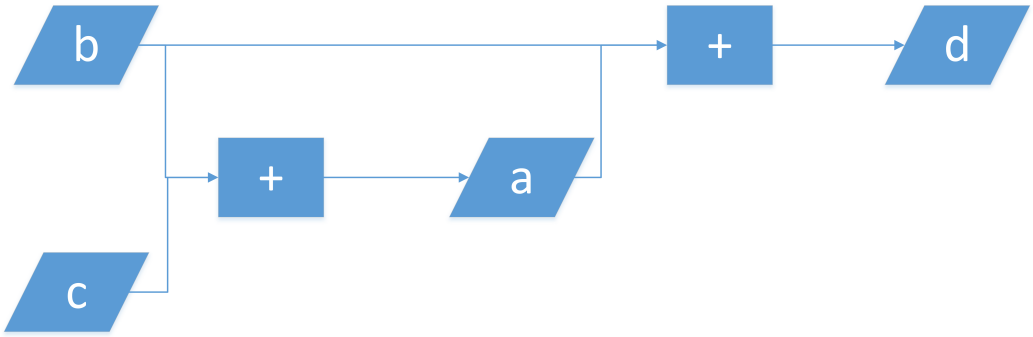

Figure 2.2: Graph of instructions

In a dataflow engine, these instructions would be registered as instructions in the dependency graph, as visualized by figure 2.2. The values of b and c would be added to the queue at application startup. B would be entered as a token twice; once for the instruction "+" which computes a and once for the instruction "+" which computes d. When the dataflow engine spins up and starts processing arguments, it sends the tokens for b and c to the first "+" instruction, which is triggered because all of its inputs are present and valid. This produces a value for variable a, which gets added to the token queue again as the first parameter for the second "+" instruction. This instruction now also has all of its inputs present, which allows it to compute the value for variable d at this point.

If at any point in the future, b or c (which should be seen as the output of other instructions not shown in the sample code) produce new values, these would be enqueued again for further processing.

2.3.3 Advantages

The key advantage of the data flow model is that only the data dependencies of the instructions decide when an instruction can be executed. Since data

flow instructions are not allowed to access or manipulate shared state, each instruction is completely isolated. This means that all dataflow instructions can be run in parallel, across different processes and even separate machines.

2.4 Conclusion

Two paradigms were presented: reactive programming and the dataflow model. Reactive programming is targeted more towards events and reactions and shines best in environments where these things are plenty, for example in user interfaces and other places where events can come from any direction. The dataflow model on its part focuses more on the parallel execution of instructions by streaming parameters to them in isolated scopes. We do however note similarities between the two, namely that they both work with an update graph that guides the data along the nodes.

3 Language

3.1 Introduction

For the purposes of this thesis, we have implemented a lightweight language called *FrDataFlow* using two different AST interpreters in Racket, supporting most of the basic constructs found in Racket. This language is based on earlier work detailed in Abelson et al. (1999). The first interpreter supports reactive patterns on top of the already existing Racket language. The second interpreter also supports the exact same language, but executes its instructions using an underlying dataflow engine. We chose to implement this custom language *FrDataFlow* (rather than a framework or library) to maintain maximum control over the inner workings and to facilitate experimentation atop the dataflow engine. The main goal was to have a reactive superset of Racket for experimental purposes during this thesis.

3.2 FrDataFlow

3.2.1 Sample program

FrDataFlow provides an environment with most Racket language constructs (defining variables, procedures, lambda, primitive operators, ...), the *lift* operator (which creates a new signal based on other signals) and some built in signals, shown in listing 3.1

```
;; A signal which emits the current seconds since 1 Jan 1970, every second
current-unix-timestamp
;; A signal which emits the current temperature from time to time
current-temp-fahrenheit
```

Listing 3.1: Built in signals

Take for example a roadside digital billboard which displays the current date and temperature. To show this information, we can derive a signal that contains the exact information that needs to be shown, using the built in signals in FrDataFlow. From the *current-unix-timestamp* signal, we can compute the date using the built in procedures *seconds->date* and *date->string*. The temperature is unfortunately in fahrenheit, so we will convert it to Celsius first. Lastly, we combine these signals into a single signal that contains the text we want to show. See listing 3.2 for the full definition of the billboard text written with FrDataFlow constructs.

```
(define (fahrenheit->celsius fahrenheit)
  (quotient (* (- fahrenheit 32) 5) 9))
(define current-temp-celsius
  (lift fahrenheit->celsius current-temp-fahrenheit))
(define current-date
  (lift seconds->date current-unix-timestamp))
(define billboard-label
  (lift
    (lambda (temperature date)
      (string-append "Temperature: " (number->string temperature) "CÂř, Date
          : " (date->string date)))
     current-temp-celsius
     current-date))
```

Listing 3.2: Billboard

This ultimately produces a signal *billboard-label* which will update every time either *current-unix-timestamp* or *current-temp-fahrenheit* produce new values. When this happens, the runtime will recalculate the values of *current-date* and *current-temp-celsius*, which in turn will trigger the update of *billboard-label*. Also note that *billboard-label* will not produce a value until both the current date and the current temperature are known.

3.2.2 Evaluation

Preprocessing

When this program is evaluated, FrDataFlow will dynamically build the signal graph by registering dependent signals as children on each signal. This means that, when a *lift* function is invoked, it will register the newly created signal as a child in every signal that is referenced. Figure 3.1 is a visualization of this graph for the example given in listing 3.2.

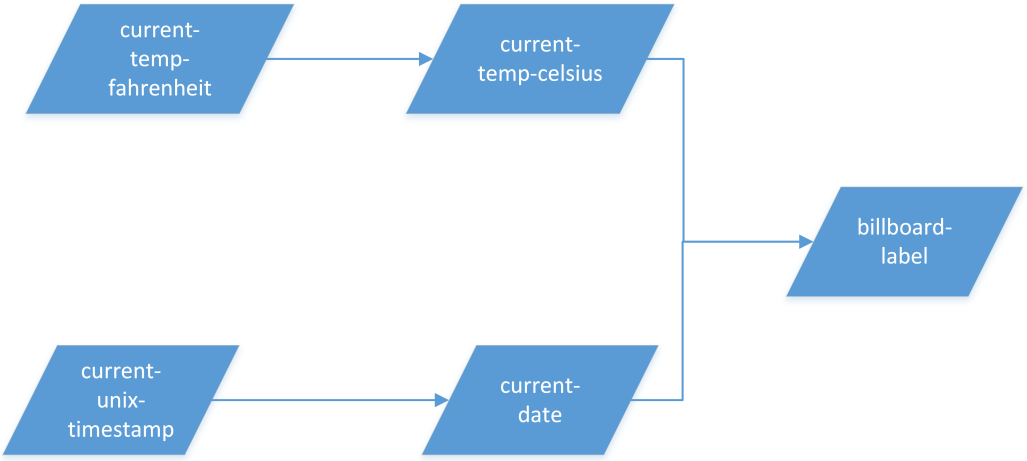

Figure 3.1: The billboard as a signal graph

Secondly, FrDataFlow will topologically sort this graph into a list, in which the signals are sorted by having the least dependencies. The position of a signal in this list indicates that it can be a child to signals that come before it, and that its children must come after it in the list. This allows FrDataFlow to simply loop over this topologically sorted list from start to finish, ensuring that signals are updated in the correct order.

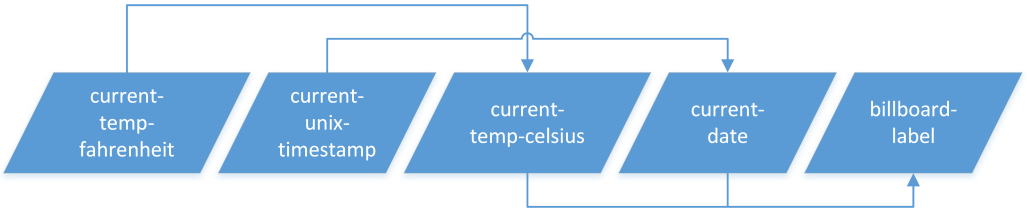

Figure 3.2: The billboard as a topologically sorted list

The update loop

To implement the update behavior, each signal has a boolean flag indicating if it is stale or not. Upon startup of the runtime, FrDataFlow initializes a never ending update loop which loops over the topologically sorted list (as shown in figure 3.2), skipping any signals which are not stale. If it encounters a stale signal, the lambda function that was provided during the creation of the signal will be called with the latest values of the parent signals. The signal is flagged as no longer being stale, and its direct children are flagged as stale immediately. Take for example an update of the current temperature. We start with a situation where the current date and temperature are already known and shown on the billboard, so the state of the signal graph looks like what is shown in figure 3.3. Note that built in signals in FrDataFlow do not have a staleness flag, because they are kept up to date in separate runtime loops.

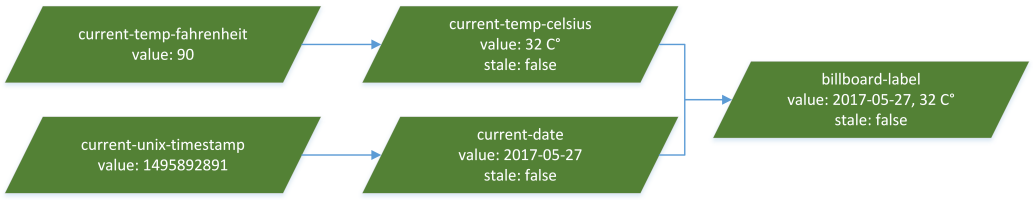

Figure 3.3: The initial state of the billboard

When the new temperature is observed, the direct children of *current-temp-fahrenheit* are flagged as stale, as shown in figure 3.4.

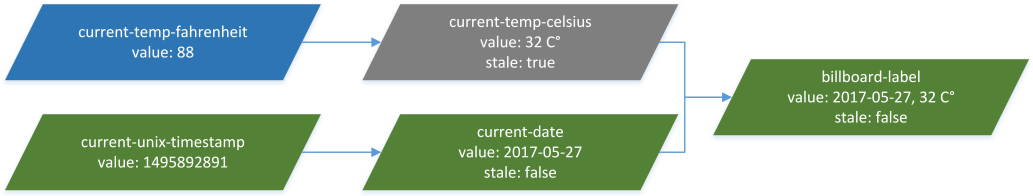

Figure 3.4: A new temperature is emitted by current-temp-fahrenheit

The update loop sees that the signal has gone stale, recalculates what its value should be using the value of *current-temp-fahrenheit* and removes the stale flag when its work is done. However, it also immediately flags *billboard-label* as stale, because it is registered as a child of *current-temp-celsius*, as shown in figure 3.5.

When the update loop moves further down the topologically sorted list, it sees that *billboard-label* is also stale now. It grabs the latest values from

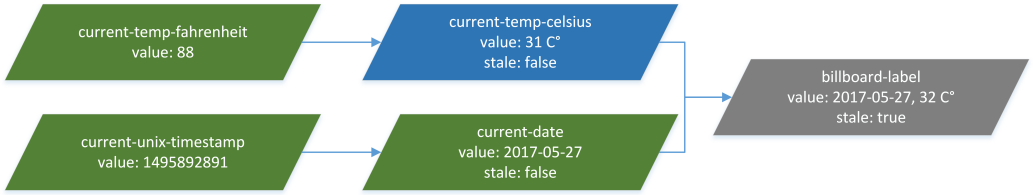

Figure 3.5: current-temp-celsius gets recalculated

current-temp-celsius and *current-date* and calls the lambda again that was used to create *billboard-label*, removing the stale flag when it is done. The final result is shown in figure 3.6. At this point, the update loop starts over again, waiting until another built in signal produces a new value.

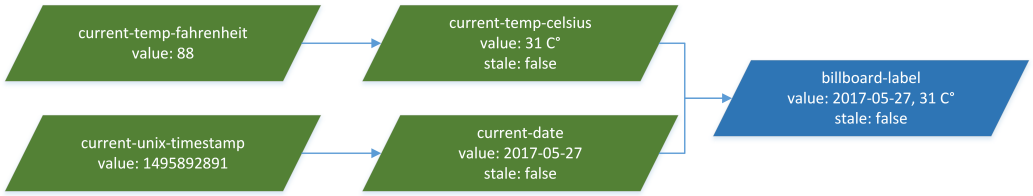

Figure 3.6: billboard-label gets recalculated

3.3 Conclusion

In this chapter the reactive language FrDataFlow was presented with samples and diagrams of its implementation. It is built as a metacircular evaluator that takes a core subset of the Racket language and extends it with some reactive concepts. FrDataFlow evaluates these expressions in a simulated runtime that forwards statements to the real underlying Racket implementation.

This language can be used to model reactive data flows and provides a built in update loop to manage the data dependencies between signals. It does this by intelligently looping over the signals while being aware of the dependencies between them. New signals can be created by deriving from other signals and a lift function which produces a single value based on the values of the parent signals.

4
Engine

5
Evaluation

6

Future work and limitations

7
Related work

8
Conclusion

A

Your Appendix

Bibliography

Abelson, H., Sussman, G. J., & Sussman, J. (1999). *Structure and Interpretation of Computer Programs* (Second ed.). London, UK, UK: McGraw-Hill Book Company.

Cooper, G. H., & Krishnamurthi, S. (2006, March). Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems* (pp. 294–308). Springer, Berlin, Heidelberg. Retrieved 2017-05-27, from `https://link.springer.com/chapter/10.1007/11693024_20` doi: 10.1007/11693024_20