Vrije Universiteit Brussel

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

Projection of Reactive Programs 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, as shown in listing 2.1.

```
a = b + c
```

Listing 2.1: A basic reactive program

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,

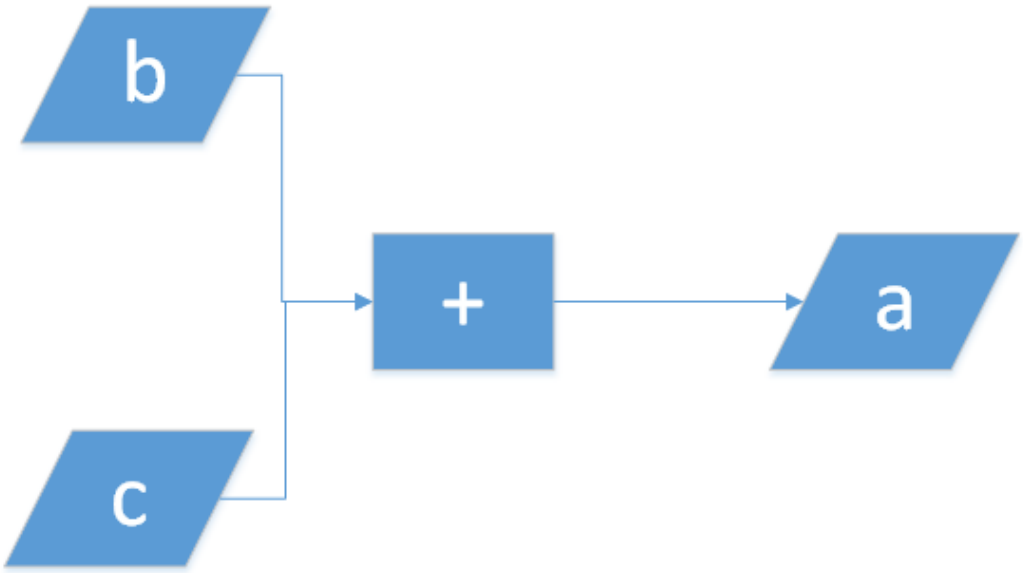

Figure 2.1: Graph of signals

at which point the value of variable a changes. See figure 2.1 for the reactive 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, as shown in listing 2.1.

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

Listing 2.2: A basic data flow program

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!

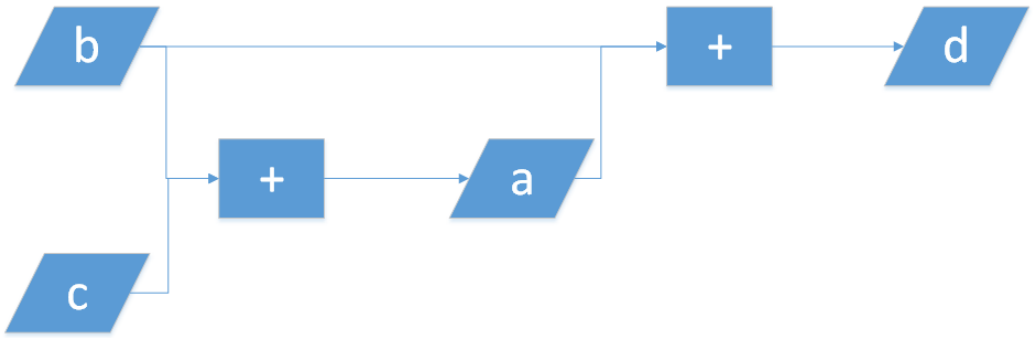

Figure 2.2: Graph of instructions

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.

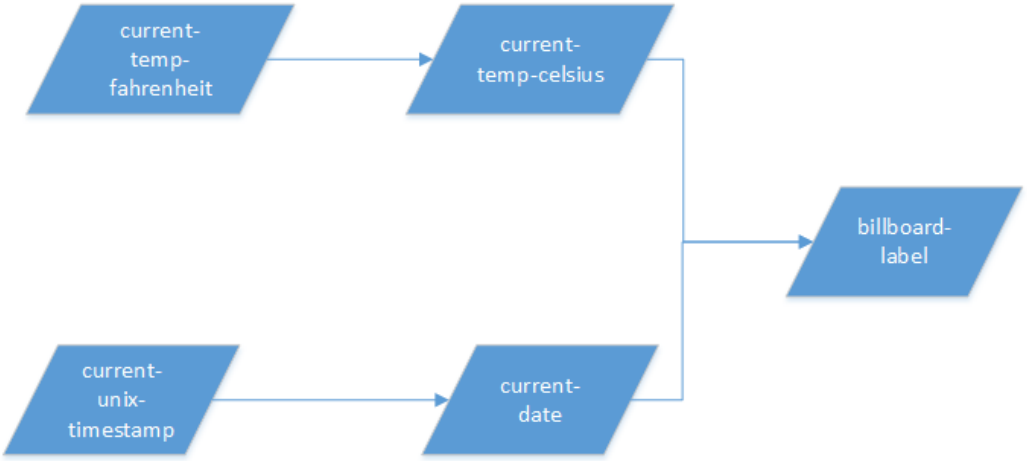

Figure 3.1: The billboard as a signal graph

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.

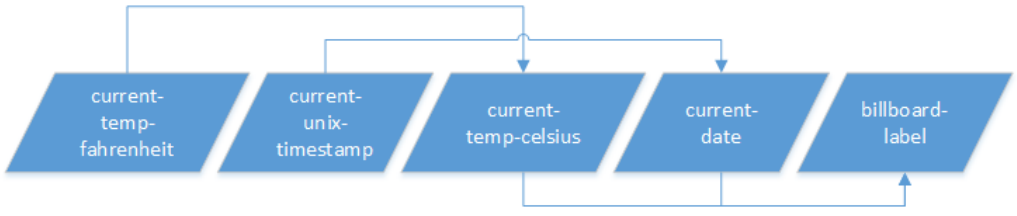

Figure 3.2: The billboard as a topologically sorted list

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.

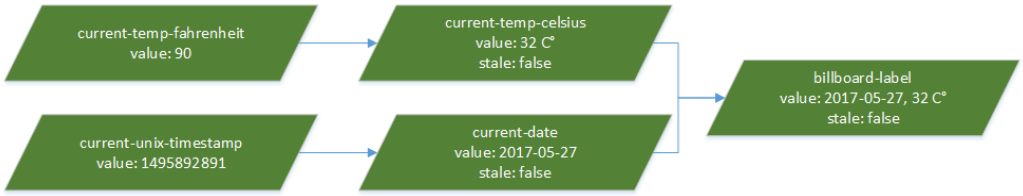

Figure 3.3: The initial state of the billboard

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

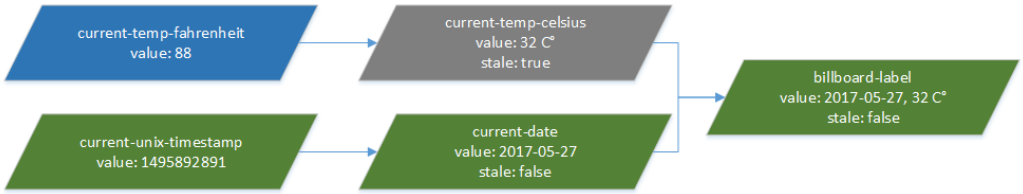

Figure 3.4: A new temperature is emitted by current-temp-fahrenheit

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

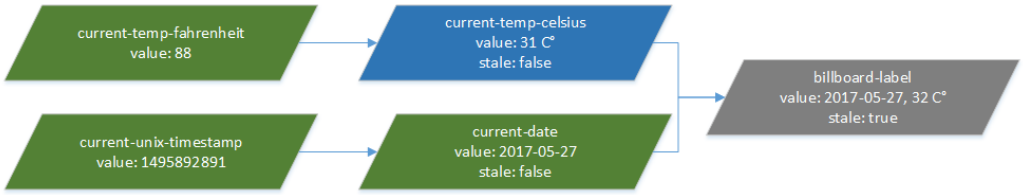

Figure 3.5: current-temp-celsius gets recalculated

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.

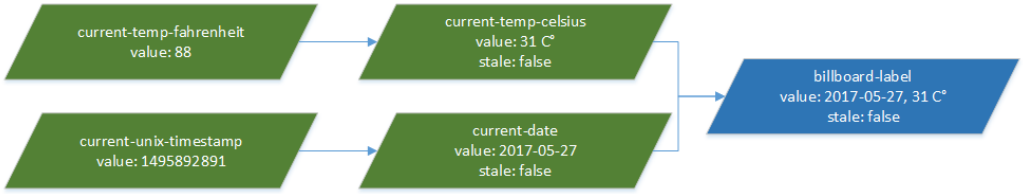

Figure 3.6: billboard-label gets recalculated

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

4.1 Introduction

In this chapter, a dataflow engine is presented based on the work detailed in Jennifer & De Meuter (2017). This engine is designed to support parallel execution of instructions by queuing all arguments to instructions and executing these in a scope agnostic way. Tagged token dataflow is an extension to the data flow model where tags are used to distinguish the execution context of tokens, i.e. multiple calls to the same instruction with different arguments. For the purposes of this thesis, a lightweight version of this engine has been implemented in Racket to facilitate the mapping process from FrDataFlow. Furthermore, we implemented a mapping layer that translates reactive signals to data flow instructions. Even though there were some mismatches that needed to be addressed, the mapping was successful.

4.2 Architecture

4.2.1 Execution of instructions

Take for example a sample program which computes the average of two numbers, as shown in listing 4.1

```
(define (average x y)
  (/ (+ x y) 2))

(average 2 6)
(average 1 5)
```

Listing 4.1: Computing the average of two numbers

This program computes the average of two numbers twice using different arguments. When this program gets evaluated in the data flow engine, the four arguments are added to the token queue, containing information about which execution context they belong to and what should happen to the result. In this example, the 2 and 6 belong to the same execution context, while 1 and 5 will belong to a different one.

Figure 4.1: The *average* instruction gets called twice

In figure 4.1 we see the arguments queued up against the *average* instruction. Colors indicate the execution context, being positioned lower and more to the left indicates their position in the token queue. At this point, the token queue consists of 2, 6, 1 and 5, wrapped in meta data.

Figure 4.2: The arguments are forwarded to the *plus* instruction

Since all arguments are present for the first invocation, the *average* instruction gets invoked, as shown in figure 4.2. This calls the plus instruction internally, which adds two new tokens to the queue: 2 and 6. Because tokens are enqueued at the end, the next thing that happens in figure 4.3 is the same step for 1 and 5. Note that tokens are processed one by one, which is

Figure 4.3: The next set of arguments are also forwarded to the *plus* instruction

Figure 4.4: Results of the *plus* instruction are passed on the *quotient* instruction

Figure 4.5: Results of the second invocation of *plus* are passed on the *quotient* instruction

not directly obvious from the diagrams. For brevity, the diagrams only show states when an instruction can be executed in the next loop.

When the *plus* instruction has completed its work, it enqueues the result as a new token again, sending it along to the *quotient* instruction. This is possible because of the information contained in the tokens: they know where to go next when instructions have completed. In figure 4.4 and 4.5 we see this process happening for both execution contexts. Note that the tags are very important here, otherwise the engine would not be aware which arguments belong to the same invocation. There is no guarantee that tokens will always be enqueued in the correct order.

Figure 4.6: These results are fed into the quotient operator

In the end, when the *quotient* instruction has also completed, its results are pushed into the token queue again, as shown by figure 4.6 and 4.7. This allows further processing of the data, but is outside the scope of this example.

Figure 4.7: The second set of results are fed into the quotient operator

To summarize, the general algorithm of the dataflow engine is to process tokens in the queue (FIFO), execute instructions whenever an execution context has all the necessary arguments present and re-enqueue the results of those instructions. Separate invocations are distinguished using tags that denote the execution context.

4.3 Mapping of reactive signals to dataflow engine

To execute our reactive signals on the dataflow engine, a translation step was required to simulate the update loop using the described mechanisms in the dataflow engine. When we represent reactive programming as a graph, the nodes are signals while the vertices denote dependencies between them. In the dataflow model, the nodes are instructions and the vertices are data being passed in to them. Therefore, our mapping layers creates data flow nodes for every signal, where the arguments passed in are the parent signal values. When a new value is produced, a token is generated for every child that is subscribed to it.

Taking our example from the Language chapter, we have the following reactive program in figure 4.8.

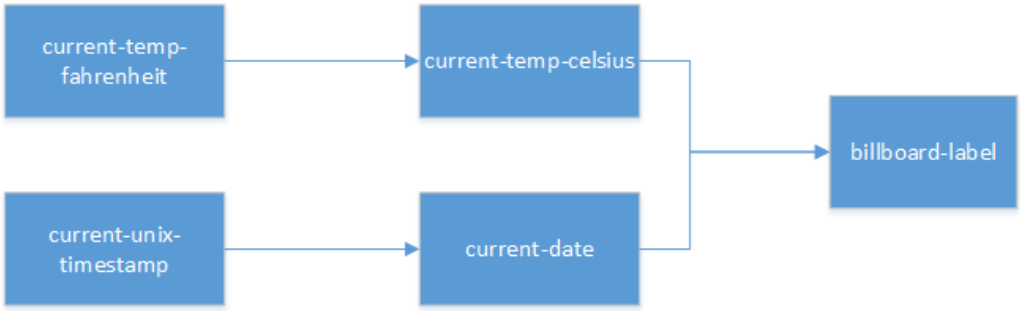

Figure 4.8: The billboard program, but the signals are now data flow nodes

This program behaves exactly the same as the reactive version, the only

difference being that is now running atop a dataflow engine.

4.3.1 Problems encountered

The consumption of arguments in the data flow model

Problem description

In reactive programs, signals with a dependency to two or more parent signals are updated whenever at least one of their parents emits a new value. This boils down to the signal functioning as a mapping of the latest values of its parents. Taking our example, this means that when the current temperature changes, the billboard will update immediately, even if the current date has not changed at all. Similarly, the billboard label updates when the date changes, even if the temperature has not emitted a new value.

In the dataflow engine, an instruction that gets executed consumes its tokens. This is problematic, because when the current temperature emits a new value, it now produces a new token on the queue. The billboard label node receives this token, but will keep waiting until the current date produces another token as well. This means that a signal does not update until ALL of its parents have emitted new values. This was the biggest mismatch between reactive programming and the data flow model, and a perfect solution was not found.

Solution description

To simulate the same behavior as reactive signals in the data flow model, we had to figure out a way to retrieve the latest values of other parent signals when a new token arrives. Although each signal has references to all of its parents, it was not advisable to just read out the latest values and generate extra tokens for the other parents as well. This would break our potential for parallelism, because it should be presumed that these dataflow nodes live on and get executed in separate processes and even machines. The whole premise of the data flow model is that nodes cannot be allowed to access outside state apart from the incoming arguments from the tokens.

The only option that remained was therefore to simulate new signal emissions for all the source signals (signals without parents that get updated by the runtime outside the update loop) whenever one of the source signals emits. Since there is no language support for filtering, throttling or dynamically manipulating the flow of data through the graph, it can be safely assumed that emitting new values from the source signals will ripple through the entire graph, causing all nodes to be updated in the same order. In prac-

tice, this meant that whenever a new temperature is sent out, the current second also pushes a value out at the same time. This means the necessary tokens are put on the queue at the same time, simulating the behavior we originally wanted: whenever one parent emits, the other parents do too.

The evaluation of all instructions in the data flow model

Problem description

The data flow engine is actually designed to execute all instructions found in the code, from the highest level abstractions to the lowest primitive operators. The mapping engine was originally designed this way to completely translate not only the signals, but also the statements found in their callbacks to data flow nodes. This meant complete buy-in into the dataflow model, in the hopes of achieving maximum parallelism. However, there is a large mismatch between the traditional Racket language and the dataflow engine: returning values. Whenever a procedure is executed in Racket, it must return a value. In the data flow model, this concept does not exist. Output from instructions is simply enqueued and is not accessible outside the dataflow runtime. Secondly, procedures in Racket carry around with them lexically scoped environments, allowing the use of closures to capture variables or other state that is not enclosed in the body of the procedure definition. Again, this violated the core principles of the dataflow engine, which meant we had to scale down the extent to which we made use of the dataflow engine.

Solution description

Instead of completely buying into the dataflow engine, we only used its model to implement our reactive signal graph. When parents emit their values, a callback is executed for the signal to update its value. This callback is actually executed inside the interpreter, using the lexical scope assigned to the procedure, without any knowledge or awareness of its presence inside the dataflow engine. While this has the downside of reduced parallelism, it does improve correctness. Of course, it's impossible to prevent mutating global state or generally causing side effects in these callbacks, but a foolproof mechanism was not the intent of this experiment.

4.4 Conclusion

In this chapter, we presented a tagged token dataflow engine. This is a runtime that enqueues arguments to instructions and executes these in parallel. When interpreting code in the FrDataFlow language, the reactive signals are mapped to dataflow nodes in the engine using a custom mapping layer. During this process, mismatches between reactive programming and the dataflow model are tackled. For example, all source signals emit new values whenever one of them has a new value, to avoid signals waiting in the dataflow engine for inputs from all their parents. Secondly, since return values are not supported in the dataflow engine, the decision was made to only model signals as dataflow instructions and leave the rest of the code execution inside the original interpreter code. In the end, we have a reactive system running atop a parallel dataflow engine, triggering updates in the correct way whenever a parent signal emits.

5
Evaluation

6 Future work and limitations

6.1 Introduction

In this chapter we present the limitations and possible improvements of FrDataFlow. The current implementation is rather limited in its functionality because we wanted to focus on the mapping layer and the possibilities of parallelization using the dataflow engine. However, multiple improvements can still be made. The setup of signals is still a rigid system in that it does not allow the dynamic creation of new signals, the filtering of values or just generally manipulating the timing of value propagation. Every signal always has to emit a value for every set of incoming values. While this already allows for a rich set of programs, these manipulations would be necessary to implement any non trivial program using FrDataFlow.

6.2 Dynamically creating signals

One of the common patterns in reactive programming is the dynamic creation of new signals based on some state or external input. In a web environment, imagine a program that searches a database while the user types keywords. This stream of text can be represented as a signal that is attached to the text input field and emits strings upon every keystroke. Then we create a

child signal which subscribes which takes these strings and launches an XMLHTTPRequest (ajax request) to fetch the data that satisfies these keywords. Every request again can be modeled as a signal which emits the data that is retrieved and then goes silent. Essentially what this program is doing is dynamically creating a new signal (XMLHTTPRequest) for every value in another signal (the keystrokes). To take this example even further, when the user types another character, we would like to discard any previously created XMLHTTPRequests because we no longer care for their results. Ideally, some sort of cancellation could happen if it is not already too late, but we definitely want to ignore any data that is retrieved based on outdated keywords. This means our dynamically added signals are not only quickly created, but also discarded.

An implementation of this pattern can be observed in the RxJs[1] library. (*Observable | RxJS API Document*, 2017). It contains the *switch* operator, which is visually represented by figure 6.1.

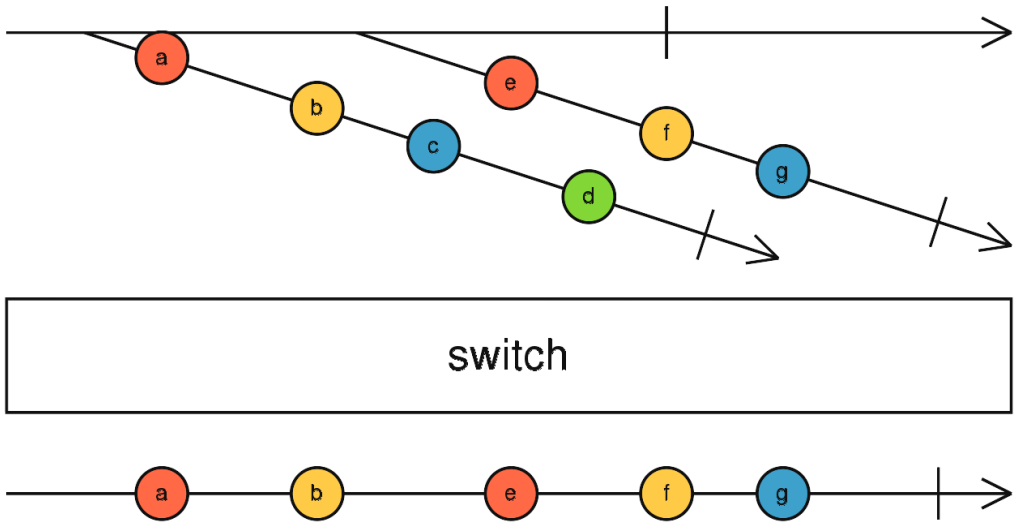

Figure 6.1: A timeline diagram of the *switch* operator in RxJs

In this figure, we can see a top horizontal arrow which represents the root signal. Values of this signal are transformed into child signals, which on their part can again emit multiple values. Since we are only interested in the latest data, we essentially *unsubscribe* from the first child signal when a new value

[1]In RxJs, a signal is known as an *observable*. When signals dynamically produce more signals, they are called *higher order observables*.

appears in the root signal, so that the sample values c and d never propagate to the end result.

In the current version of FrDataFlow, it is unfortunately not possible to create new signals at runtime. A possible improvement would be to allow this, which would mean the signal would have to be added to the topologically sorted signals and be translated to a dataflow node. This would also mean that any existing dataflow nodes upon which this new signal depends would have to be updated to also generate tokens for this new node. Similarly, upon destroying signals, this node and all the references to it would have to be updated.

6.2.1 Filtering

Another common practice is filtering of signals, only letting values through which satisfy a certain predicate. Our current implementation of FrDataFlow models signals as a mapping function which takes a set of inputs and produces a new value for every updated set. Filtering would introduce the possibility of not propagating a value at all if certain criteria are not met. In essence, a signal would no longer provide a lambda that has to return a value, but a lambda that can push values into a type of sink when it wants to. This refactoring would allow for a whole category of powerful constructs, such as taking only the first n values, throttling signals, etc.

An example visualization of the filter function in RxJs can be seen in figure 6.2.

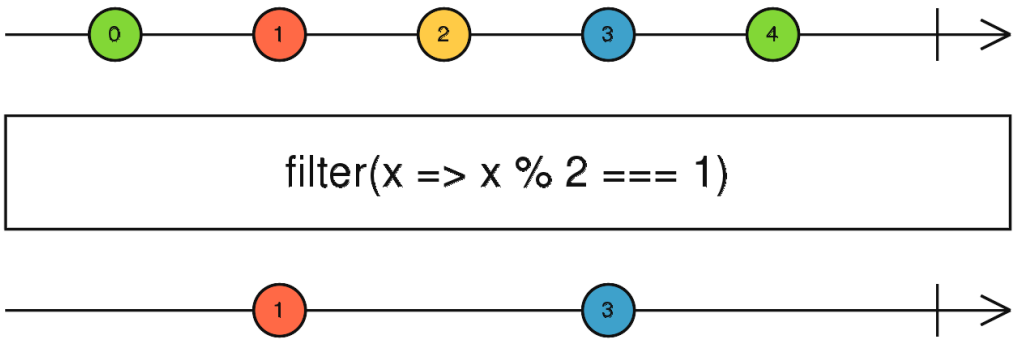

Figure 6.2: A timeline diagram of the *filter* operator in RxJs

We see at the top a signal which produces a stream of numbers. If we create a new signal that filters these numbers using the filter function, we get

a new stream of numbers that only contains those that satisfied the predicate. When the number 0 is pushed into the signal lambda, it does not propagate it into the sink of the child signal, which means the value does not go through.

To implement this pattern in our mapping layer, we would have to ensure that dataflow nodes don't always have to produce new tokens when they are invoked, and a way for nodes to indicate when this should happen or not. However, there is no technical reason why the dataflow engine should not be able to handle this: it is simply the presence of tokens that determines which nodes are invoked, so not emitting these tokens would essentially implement the filtering behavior we have described.

6.3 Conclusion

In this chapter we have shown how FrDataFlow is still a basic language that does not support some powerful concepts such as the dynamic creation of signals or the filtering of values. If these concepts were to be introduced, they would allow for an implementation of most of the techniques commonly found in other reactive libraries or languages.

This initial version of FrDataFlow has focused mostly on the practical mapping layer from reactive programming to the dataflow model. There are however no immediate technical shortcomings that should prevent us from implementing the modifications presented in this chapter.

7 Related work

7.1 Introduction

In this chapter, we introduce a few related works and papers that align with concepts described in this dissertation. The language Elm for example describes in detail the challenges of functional reactive programming, tackling many of the same problems and describing similar solutions as encountered in FrDataFlow. Despite the similarities, the difference in platforms (Elm has to adhere to the limitations of the JavaScript execution engine in browsers, while FrDataFlow is constrained to the limits of the Racket runtime) has a significant impact on the potential to parallelization.

7.2 Reactive programming

7.2.1 Elm

In 2012, Evan Czaplicki wrote his dissertation on Elm (Czaplicki, 2012), a functional reactive programming language for the web. In this paper, he presents a new language *Elm* that supports concurrent functional programming for the web. Elm is compiled down to JavaScript using a compiler written in Haskell and promises to produce no runtime errors when everything compiles. Most of the reactive programming concepts in FrDataFlow

were actually heavily inspired by Elm, such as *signals* and the *lift* function. Another concept inspired by Elm is signal graphs: reactive primitives as nodes and value flows as edges. Of course, source signals are a bit different in the web. For example, Elm provides source signals that represent mouse clicks, mouse positions or generally any kind of DOM event which is captured by the standard library. In the isolated, experimental little world of FrDataFlow, primitive source signals other than the current time were hard to come by.

Order of events

Looking at a signal graph, it is easy to imagine the parallel execution of the nodes. However, the order of events is very important. Take for example the signal graph shown in figure 7.1.

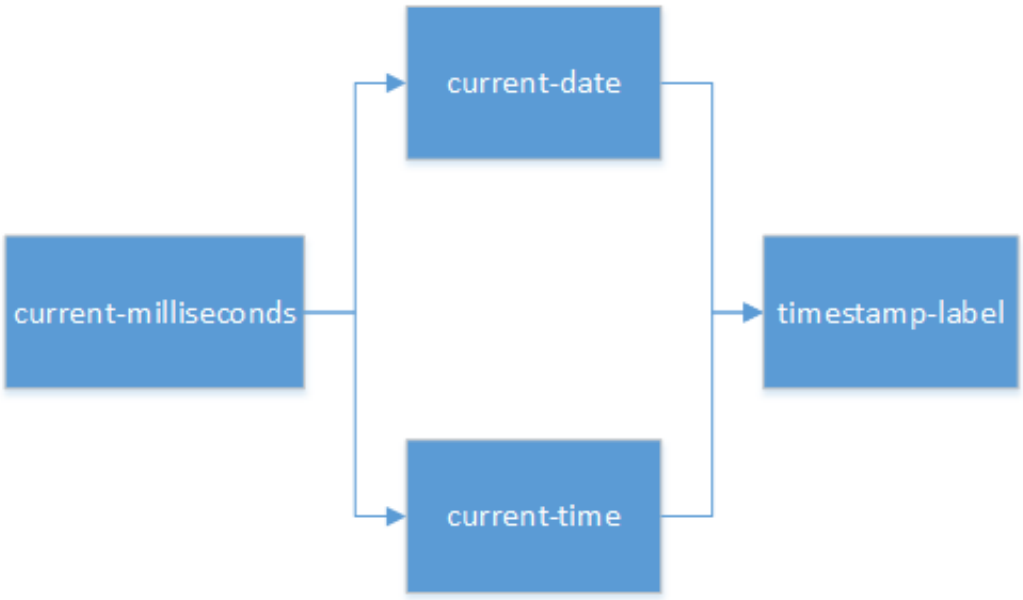

Figure 7.1: The current timestamp is based on two different computations of *current-milliseconds*

If a new value appears for current-milliseconds (the number of milliseconds since 1 Jan 1970), we compute the current date and current time separately, to combine these two data points into a single label called *timestamp-label*. Imagine now that the computation of the current time takes significantly longer than the computation of the current date. In fact, we can contemplate that the derivation of the time would actually need to derive the date first, and then use the remaining milliseconds from midnight to

determine the time. Of course, depending on the date and the timezone, this could result in different clock times. This leads us to conclude that the calculation of *current-time* is slower than *current-date*.

Now imagine that these computations are run in parallel and that their outputs are piped to the timestamp label as fast as the hardware allows it. If the date is so much faster to process, it becomes possible that the timestamp label starts receiving more than one *current-date* for every *current-time*, leading to situations where the date and time that shown do not both trace back to the same value of *current-milliseconds* they were derived from.

In Elm, this problem was fixed by the introduction of a global event dispatcher, which imposes a few constraints on the signal graph update loop:

- Signals with more than one parent must wait for all parents to produce a value before recomputation happens. This is the same approach taken in FrDataFlow.
- When source signals emit a new value, all nodes receive that event and pass forward a "nothing changed" notification. In FrDataFlow, the approach is to simply push the current value.

Parallelization

In Elm, the constraints of the platform are slightly different. The JavaScript runtime does not really provide native parallelization[1], so while the language itself could perfectly support parallelism, unfortunately its platform does not.

Elm does provide some workarounds for when the synchronous nature of its node processing causes performance bottlenecks, namely asynchronous updates. This keyword decouples a subset of the graph from the main graph and allows it to update independently, avoiding the situation where it would have to wait for long running synchronous updates. However, this is still purely a data correctness and timing feature and unfortunately does not tackle the lack of parallelization support.

In fact, the paper does mention a possible solution to run Elm programs in parallel: if closures can be avoided somehow (for example by compiling to an intermediate language which explicitly lists the used captured variables)

[1] Web workers exist, which were designed offload work in background threads. However, they come with two rather expensive limitations, the first being that messages between workers must be primitive data types, so there is no support for passing along functions with closures, only simple messages such as strings or numbers. Secondly, these web workers directly map to operating system threads, making them quite expensive to set up and tear down. The end result is that these workers are not a good fit for the small, atomic computations we are dealing with in reactive nodes.

and if functions are passed as strings (and then dynamically interpreted inside the workers), web workers could technically provide a parallel execution mechanism. It was not considered wortwhile though, because of the amount of overhead to orchestrate this and the possible security ramifications.

7.3 Dataflow Model

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. doi: 10.1007/11693024_20

Czaplicki, E. (2012). *Elm: Concurrent frp for functional guis.*

Jennifer, M. S. J. D. K., & De Meuter, B. S. W. (2017). An Extensible Virtual Machine Design for the Execution of High-level Languages on Tagged-token Dataflow Machines.

Observable / RxJS API Document. (2017, June). Retrieved 2017-06-01, from `http://reactivex.io/rxjs/class/es6/Observable.js~Observable.html`