VRIJE
UNIVERSITEIT
BRUSSEL

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

PROJECTION OF REACTIVE PROGRAMS ONTO DATAFLOW ENGINES

ALEXANDER MOERMAN

Academic year 2017 - 2018

Promotor: Prof. Dr. Wolfgang De Meuter
Advisors: Mathijs Saey, Florian Myter and Thierry Renaux
Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proef ingediend met het oog op het behalen van de graad van
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

PROJECTION OF REACTIVE PROGRAMS ONTO DATAFLOW ENGINES

ALEXANDER MOERMAN

Academiejaar 2016-2017

Promotor: Prof. Dr. Wolfgang De Meuter
Begeleiders: Mathijs Saey, Florian Myter en Thierry Renaux
Wetenschappen en Bio-ingenieurswetenschappen

Abstract

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

1

Introduction

Reactive programming is becoming increasingly popular. Over the past decade, we have observed a slow but steady shift away from traditional imperative paradigms towards other, more declarative ones. The technology landscape is moving, and with it the software development world. Web and mobile applications, the *internet of things* and now virtual reality software are dominating the conversation, and they all have one thing in common: they are driven by events. Events, callbacks and asynchronous computation have always been challenging. Since callbacks are inherently not composable and do not provide a unified interface, one quickly arrives at deeply nested callbacks to perform any meaningful task, these nested structures are colloquially known as *callback hell*. Furthermore, propagating and eventually surfacing errors requires manual wiring of these errors up the call chain, resulting in repetitive and difficult to maintain code.

Reactive programming solves these problems by putting events at the forefront as first class citizens under one unified interface, reifying them as time varying values and making them composable using declarative operators. It provides a single interface over I/O, events and any asynchronous computation, switching the mindset from state through imperative modifications to state distilled from declarative derivations of events.

1.1 Problem statement

Timing and efficiency is critical and desired for reactive systems (e.g. user interfaces, robotics, etc.). Thus far the majority of reactive programming implementations have been single threaded and don't utilise the maximum potential of their underlying host. Slowly but surely there are parallel approaches popping up which focus on thread based or actor based concurrency [9], but none have tried mapping these reactive programs onto something that was designed to run parallel from the beginning: the dataflow execution model. This execution model allows instruction invocation whenever the inputs are available (contrary to the sequential models such as the Von Neumann model where instructions are executed one after another) and does not have global state, which allows it to exploit all of the available parallelism in a given input program.

Reactive programming nor the dataflow execution model are new technologies, both go back to at least the 1980s [7, 12]. While they try to solve different problems, they do share an interesting common trait: the evaluation models of both use directed graphs to orchestrate data flowing through its applications.

The core principle of reactive programming is making events first class citizens, called signals: they can be listened to, transformed or even composed with other signals, ultimately building a graph of nodes that represent these signals and the data dependencies between them. When the application executes, data courses through this graph via the *reactions* of the nodes when a signal is fired. One could say that the system *reacts* to every signal.

The dataflow execution model on the other hand has a different purpose: parallelizing primitive instructions as much as possible by tracking the flow of data through them and invoking instructions as soon as their inputs are present. Unrelated instructions, which are not dependent on common data, can be executed in parallel, while related instructions are only invoked when their data dependencies are satisfied. These dependencies between instructions again form a graph. This time however, nodes in the graph represent instructions, not signals. Dependencies in this graph simply indicate that one instruction takes as input the output of another.

So while reactive programming uses a graph of nodes which represent signals, the dataflow execution model has a graph which represents instructions. When executed, data flows through these graphs and the output of a node is forwarded to nodes further down the graph. Conceptually, nodes in these graphs mean different things in the two models, but the way that they behave is undeniably similar.

The goal of this thesis is to explore the possible benefits of running a re-

active language on top of a dataflow engine, more specifically the advantages with regards to parallel execution. We propose that the update mechanism necessary to support a reactive language can be completely implemented on top of a dataflow engine, with the aim to parallelize the updating of separate nodes in the reactive graph. This would allow us to scale a reactive program horizontally, exploiting the maximum amount of parallelism possible. The purpose of this thesis is therefore to investigate the parallelization of reactive programs by using the dataflow execution model as a platform.

1.2 Contributions

In order to test our hypothesis, we present a new reactive programming language *FrDataFlow*, based on Racket with a few extra features to natively support reactive programming. Expressions in this language are evaluated by our own interpreter, which builds the aforementioned graph in the background and keeps it up to date. Rather than implementing our own update mechanism however, the reactive graph is translated to instructions in a dataflow engine implementation as described in *An Extensible Virtual Machine Design for the Execution of High-level Languages on Tagged-token Dataflow Machines* [10]. This is the core mechanism that will orchestrate the events and keep nodes in the graph up to date.

Furthermore, we evaluate this approach by comparing it to an implementation of the same language that does not run on a dataflow engine, but applies a more traditional evaluation model, as described in FrTime [4]. Benchmarks are provided for both latency and throughput for both implementations. Lastly, we also investigate the scaling possibilities of FrDataFlow by actually running it across multiple cores simultaneously.

1.3 Structure

This dissertation is structured to provide the reader with sufficient context before the actual research and evaluation is discussed. To this end, chapter 2 provides a general background with regards to both reactive programming and the dataflow execution model. It succinctly covers the problems they want to solve, how they attempt to do so, the advantages of these approaches and a short example illustrating how they work.

In chapter 3, our custom language FrDataFlow is presented, with an elaborate sample program displaying most of the features supported by the language. The evaluation of this example by our interpreter is then visually

detailed with diagrams and corresponding explanations.

The dataflow engine is discussed in chapter 4. This is an implementation of the dataflow execution model that powers the update mechanism in FrDataFlow. A short sample is provided to show how instructions are invoked in this engine, and how output from one instruction gets forwarded to the input argument of another. Next, the main research topic of this dissertation is presented: the mapping algorithm. This is the translation we make from the graph in reactive programming to a new graph of instructions for the dataflow engine. Encountered obstacles and corresponding workarounds are discussed.

Chapter 5 details the evaluation of our research. We investigate the consequences of running a reactive system on top of a dataflow engine with regards to efficiency. Furthermore, we discuss the benefits and downsides of running reactive programs in parallel in terms of latency and throughput of values.

In chapter 6, we summarize the boundaries at which this research has drawn a line and discuss possible future improvements or extensions to our system. We do this by recognizing features provided in other reactive languages or frameworks, that are not present in FrDataFlow.

Related research topics are discussed in chapter 7. We took great inspiration from ambitious works such as Elm and FrTime, which we compare with FrDataFlow in terms of design and evaluation.

Finally, we conclude with chapter 8, which summarizes our findings of this research. We revisit the original problem statement and summarize our contributions to this space.

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 programming paradigm allowing the declarative specification of event-driven code. It allows programmers to combine and derive events from a set of event sources (e.g. user interactions, etc.). This deviates from the traditional callback-based approach where events are dealt with using the observer pattern. 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 [4].

2.2.1 Example

The canonical metaphor for Reactive Programming is spreadsheets, which automatically recompute the value of a cell if one of the cells on which it depends changes. 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, at which point the value of variable a changes. The language runtime keeps track of the flow of data by constructing a dependency graph where the nodes are signals and the operations create edges between the nodes. See

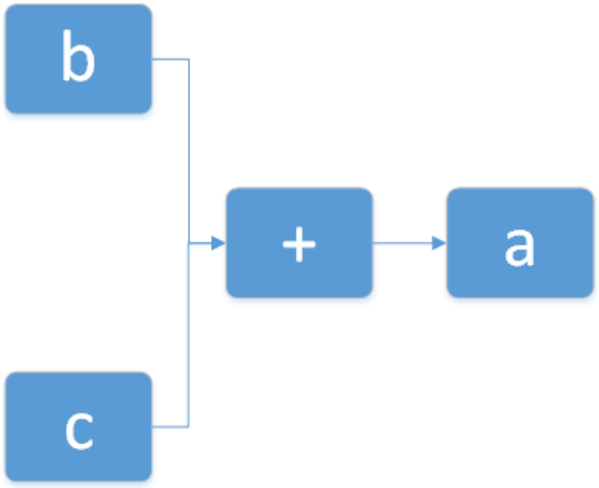

Figure 2.1: Graph of signals

figure 2.1 for the reactive graph of the sample code in listing 2.1. While the application is running, the runtime will traverse this graph every time a source node emits a new value, rippling the effect of that change across the graph. In our example, 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

Signals can be described as "values over time", in contrast with a variable which only holds its latest value, revealing no information about the time when that last value was produced or what caused the variable to be assigned a new value to begin with. Signals can be used to model time-varying values, typically found in event-driven applications:

- 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
- 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 [8] 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 operands have been provided. Contrary to imperative programming, instructions are not invoked by a program counter, but rather whenever all of the operands are present.

The execution of a program in the dataflow model can be represented as a direct graph of nodes where each node represents an instruction and each edge is data traveling between instructions. It is up to the dataflow engine to orchestrate inputs so that instructions are invoked when all of the input data is present. Whenever an instruction is invoked, the output is sent through to all connected instructions which depend on it.

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!

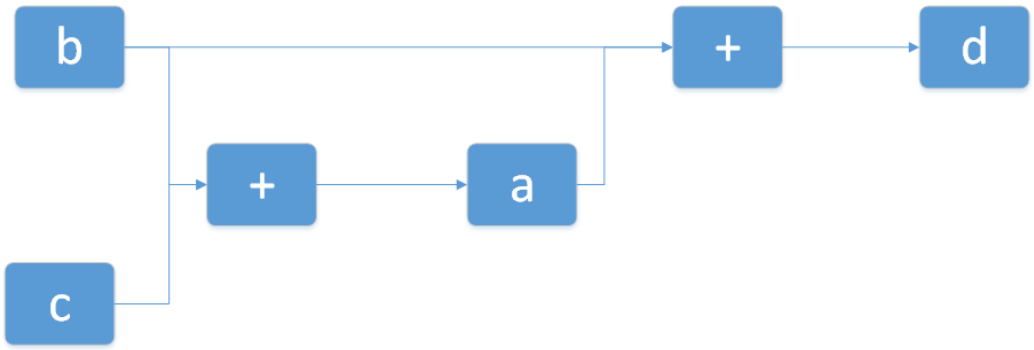

Figure 2.2: Graph of instructions

In a dataflow engine, these instructions are registered as nodes in the dependency graph, as visualized by figure 2.2. What really happens is that the values of b and c are added as tokens to the processing pipeline at application startup. The value of b is added as a token twice; once for the "+" instruction which computes the value of a and once for the "+" instruction which computes the value of d. When the dataflow engine spins up and starts processing data, it sends the tokens for b and c to the first "+" instruction, which is triggered because all of its inputs are present. This consumes the input tokens of b and c and produces a value for a, which gets added as a token to the processing pipeline again as the first operand 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 in the pipeline for further processing.

2.3.3 Tagged Token DataFlow

The dataflow model has different possible implementations, in this dissertation we solely focus on *tagged token dataflow* [2]. In this version, instruction arguments, i.e. tokens, are tagged with meta data to provide the engine with the necessary information to orchestrate the data flows and also to provide the possibility of making the graph reentrant by using execution contexts. For example, when calling a function multiple times the engine has to ensure these calls are separated correctly when invoking the instructions. A token consists of the following parts, as shown in figure 2.3:

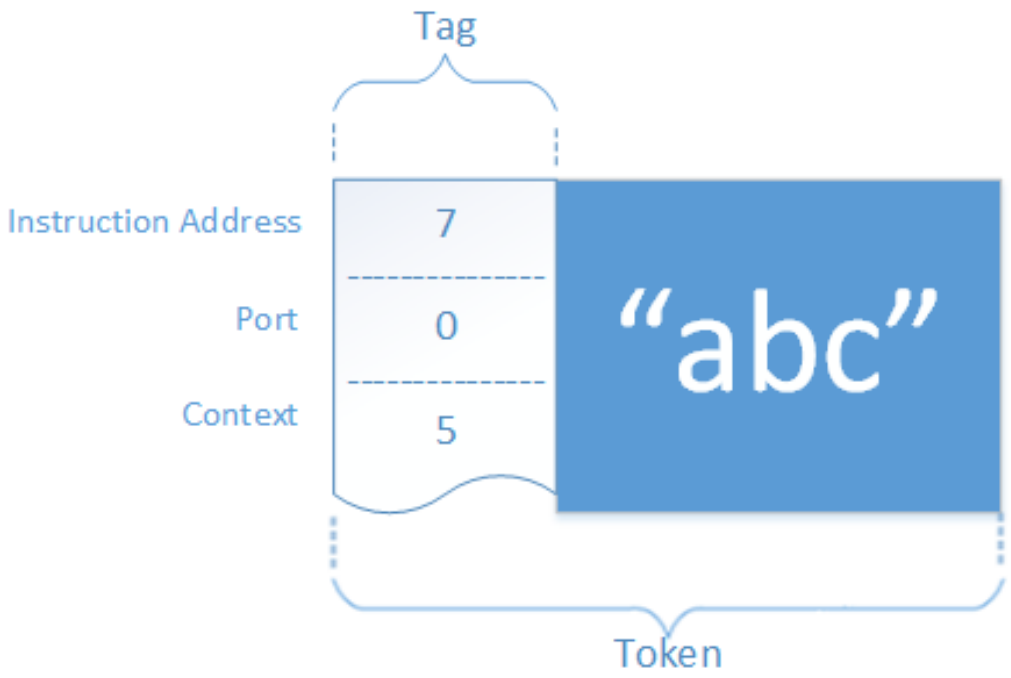

Figure 2.3: A tagged token

The value
Also called the datum, in this example "abc". This is the required data to invoke the instruction.

The instruction address
Uniquely identifies the instruction to be invoked

The port
Uniquely identifies the position of the token, should an instruction need

multiple tokens for a single invocation. For example an instruction with two inputs would require two tokens be present to invoke the instruction: one token with port 0 and one with port 1.

The execution context
This allows an arbitrary part of the instruction graph to be made reentrant. For example, it can uniquely identify a function call and distinguish instruction invocations from different function calls.

2.3.4 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.

A large difference between both is that the dataflow model 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. Dataflow execution models will also consume inputs when instructions are invoked, while reactive programs do not. On the contrary, signals reuse earlier values when only one new input is provided to compute a new result.

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 operands 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 its 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 a small subset of language features found in Racket. The first interpreter is implemented in the style of the metacircular interpreter found in *Structure and Interpretation of Computer Programs* [1] and supports reactive patterns on top of the already existing Racket language. The second interpreter also supports the exact same language, but executes its instructions atop a 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) "C°, 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 without the dataflow engine

For the initial version of FrDataFlow, the metacircular interpreter in Racket was extended to natively support signals. Note that this does **not** include any mapping to the dataflow engine yet. How the example program is evaluated and how the signals are kept up to date by this initial version of the runtime will now be described in detail.

Preprocessing

When this program is evaluated, FrDataFlow will first 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. It will also store the lambda that is provided to the lift operator as the update function for that signal. Figure 3.1 is a visualization of this graph for the example given in listing 3.2.

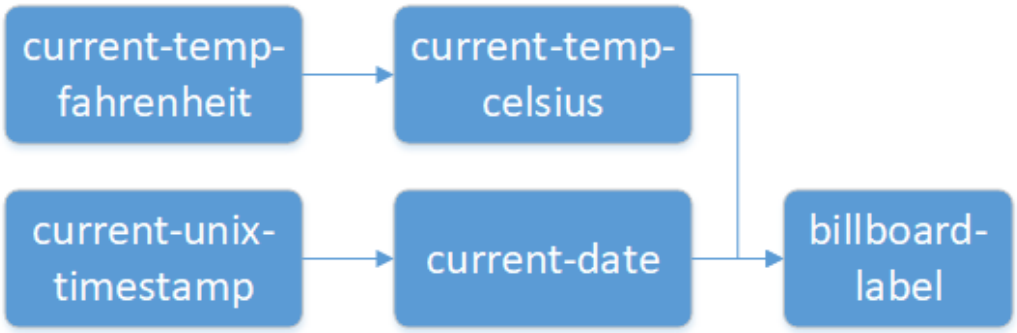

Figure 3.1: The billboard as a signal graph

Secondly, FrDataFlow will topologically sort this graph into a list, which means each signal appears later in the list than the signals it depends on. In other words, 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.

The update loop

To implement the update behavior, each signal has a boolean flag indicating if it is stale or not. A signal is considered stale if (at least) one its parents has emitted a new value, but has not been updated yet. Upon startup of the runtime, FrDataFlow initializes a never ending update loop which

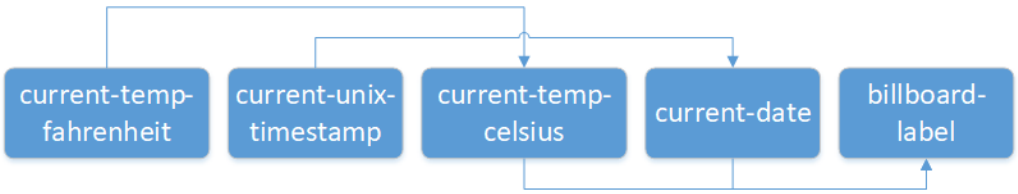

Figure 3.2: The billboard as a topologically sorted list

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 update 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 (i.e. source signals) in FrDataFlow do not have a staleness flag, because they are kept up to date in separate background loops. For example, a standalone infinite loop keeps the *current-seconds* signal up to date by updating its value every second.

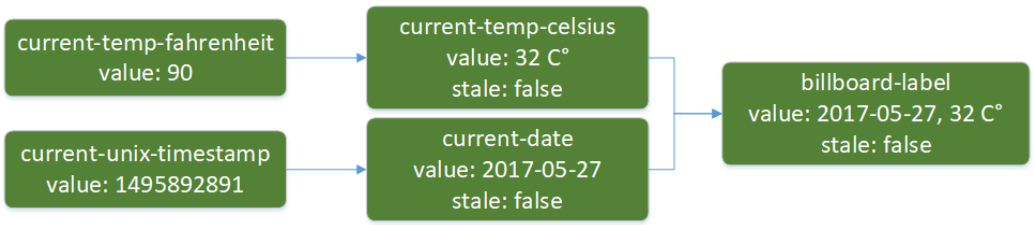

Figure 3.3: The initial state of the billboard. **Green = Not stale**

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

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 stale now. It grabs the latest values from *current-temp-celsius* and *current-date* and calls the lambda that was used to create *billboard-label*, setting the stale flag to false when it is done. The final result is shown in figure 3.6. At this point, the update loop starts over again,

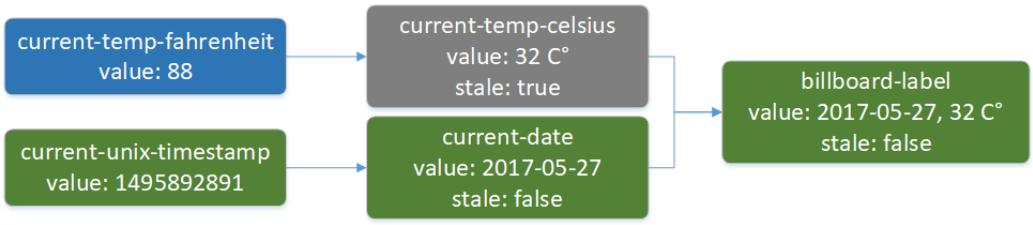

Figure 3.4: A new temperature is emitted by current-temp-fahrenheit. **Blue = New value and not stale**, Grey = Stale

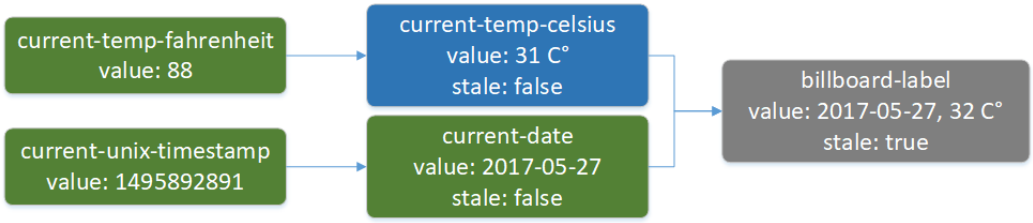

Figure 3.5: current-temp-celsius gets recalculated

repeating ad infinitum. Every time, the loop checks if a signal is stale and repeats the update process as described above. If no signals are stale, it keeps looping until that changes.

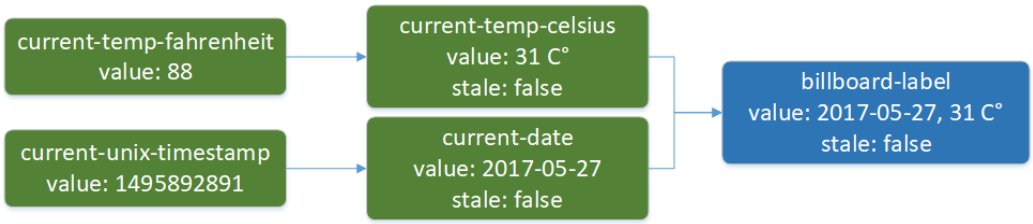

Figure 3.6: billboard-label gets recalculated

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
DataFlow Engine

4.1 Introduction

In this chapter, a dataflow engine is presented based on the work detailed in *An Extensible Virtual Machine Design for the Execution of High-Level Languages on Tagged-Token Dataflow Machines* [10].

This engine is designed to support parallel execution of instructions by queuing all operands 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 invocations of the same instruction with different execution contexts, for example calling the same function twice (See section 2.3.3). 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, reactive programs can now be expressed in FrDataFlow and evaluated atop the dataflow engine.

4.2 Architecture

4.2.1 Components

The dataflow engine consists of four core components:

The token queue
This contains the tokens that need to be processed. A token has a value and meta data about which instruction it needs to be sent to.

The matching memory
This stores the operands per instruction and per context. As long as not all inputs are present and the instruction is not ready to be invoked, these partial sets of inputs are kept here.

The execution unit
This unit will actually invoke the instructions with its operands

The interaction between these four components can be seen in figure 4.1. What the dataflow engine essentially does is continuously pull tokens from the queue, send it through the matching memory which calls the execution unit. This will invoke the instruction with its operands, wrap the result in a token and put it on the queue again.

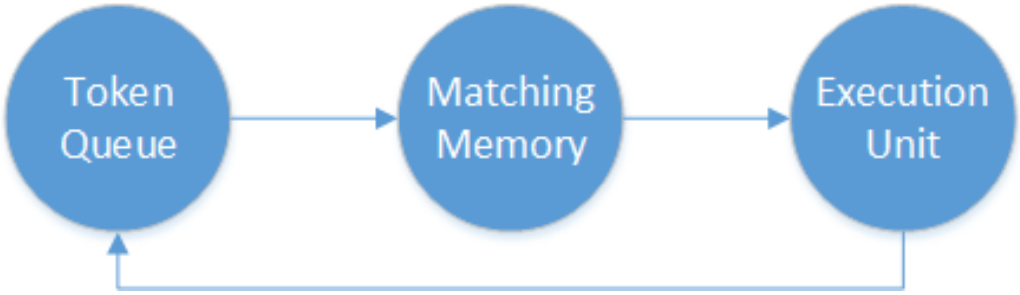

Figure 4.1: The interaction between the four components of the dataflow engine

For clarity, figure 4.2 is repeated here (from the Background chapter) to reiterate the structure of a token.

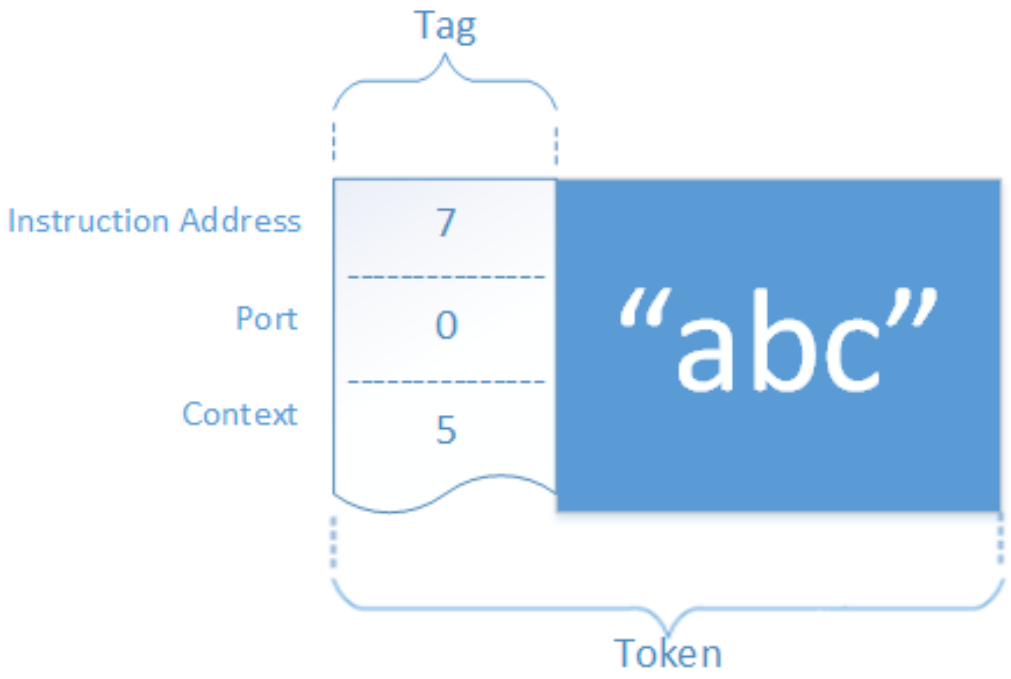

Figure 4.2: A tagged token

4.2.2 Example program with evaluation by the dataflow engine

Take for example a non reactive 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 and the 2s from the division by two are added to the token queue, containing information about which execution context they belong to

and what should happen with 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 4.3: The corresponding data flow nodes for two calls to the *average* function. Each color represents a different execution context.

In figure 4.3 we see the arguments queued up against the *plus* instruction. Colors indicate the execution context. At this point, the token queue consists of 2, 6, 1, 5 and two 2s. These values are all wrapped in a token containing meta data, i.e. the tag.

Figure 4.4: The results of the *plus* instruction are forwarded to the *divide* instruction

Since all inputs are present for the first invocation, the *plus* instruction gets invoked, as shown in figure 4.4. This adds a new token to the queue: 8. This result will serve as the input of the *divide* instruction. The next thing that happens in figure 4.5 is the same step for 1 and 5.

Figure 4.5: The second invocation of the *plus* instruction finishes. All inputs are now present for the *divide* instruction for both execution contexts.

Figure 4.6: The first *divide* instruction is invoked. Its result is put on the token queue again.

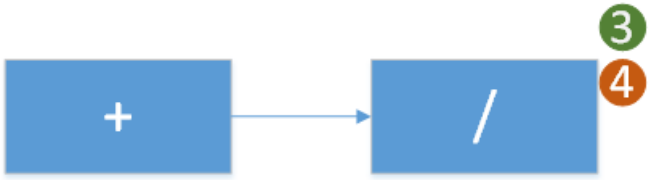

Figure 4.7: The second *divide* instruction is invoked. Its result is put on the token queue again.

Note that the tags are very important here, otherwise the engine would not be aware which inputs belong to the same invocation. There is no guarantee that tokens will always be enqueued in the correct order.

In the end, when the *divide* 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.

To summarize, the dataflow engine processes tokens in the queue, invokes instructions whenever all of its inputs in one execution context are present and re-enqueues the results of those invocations. Separate function calls 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.

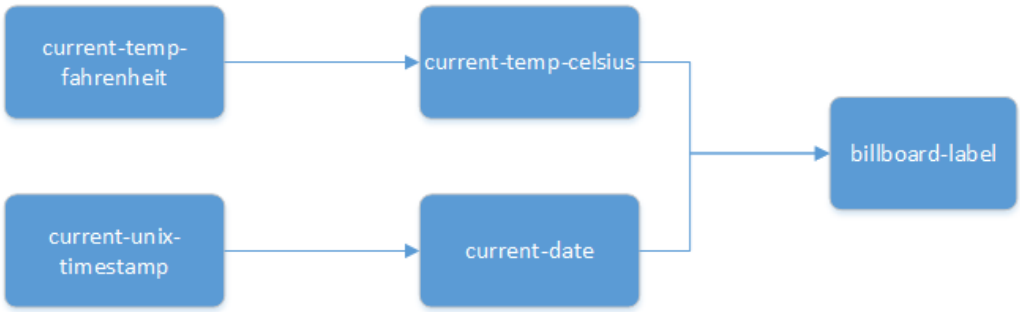

Figure 4.8: The billboard program, but the signals are now data flow nodes

This program behaves exactly the same as the reactive version, the only difference being that it is now running atop a dataflow engine.

4.3.1 Updating signals on a dataflow runtime

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 a sense, signals remember the latest values from each parent and reuse them whenever a new value from one of the parent comes in.

In the dataflow engine, an instruction that gets invoked consumes its tokens. Once the instruction has been invoked, the tokens are thrown away. This is problematic, because when the current temperature emits a new value, it produces a new token. The dataflow engine will assign this token to the billboard label node, but will keep waiting until the current date produces another token as well before invoking the billboard label again. This means that a signal which is now mapped as a dataflow node does not update until ALL of its parents have emitted new values. This is the biggest mismatch between the reactive model and the data flow model.

Simulating the behavior of reactive signals in the data flow model requires 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 is not advisable to just read out the latest values and generate extra tokens for the other parents as well. This is because a strict requirement of the dataflow model is that nodes must be processed in isolation and cannot access shared state, which is what retrieving the latest values of the parents would amount to. This would break the potential for parallelism, because it should be presumed that these dataflow nodes live in and are invoked on separate processes and even machines. The whole premise of the dataflow model is that nodes are only allowed to access the incoming data from the tokens and nothing else.

We propose to simulate new signal emissions for all of the *source signals* whenever one of them emits a new value. These are signals without parents that get updated by the runtime outside of the update loop. 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 practice, this means that whenever a new temperature is sent out, the current seconds also pushes a value out at the same time. The necessary tokens for all of the source signals are put on the queue at the same time, simulating the required behavior: whenever one parent emits, the other parents do too. This results in the desired behavior: whenever one parent signal emits, the other parents do too so that all the children are recomputed, without having to memorize any of their parents' values.

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. This is done by making 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

5.1 Introduction

In this chapter, we investigate the performance characteristics of the two runtimes we created for FrDataFlow. We will explore different types of reactive programs and how they behave under high loads of data, comparing them on a few metrics. More specifically, we will focus on the following metrics:

Latency
: The time it takes for a value to propagate through the entire graph of signals

Throughput
: The amount of values that can propagate through the entire graph within a certain time frame.

Scaling
: The amount of performance that can be gained by executing the program in parallel

These objective measurements can provide us with meaningful insight as to how efficient both interpreters are, and what the effects of parallelization are for our benchmarks.

5.2 Topologies

Before we can start running benchmarks, we must first define the types of programs that we are interested in profiling. In *Optimizing Distributed REScala* [6], three topologies are suggested that sufficiently represent any reactive program for the purposes of evaluating performance.

5.2.1 Linear

This model is the simplest one: we have a graph of signals where each signal only has one parent and one child. When the source signal emits a new value, it simply propagates through the graph in one linear motion and stops at the end.

Figure 5.1: A linear topology: each signal has one parent and one child

5.2.2 Fan out

In this topology, our graph immediately splits up into many children, as shown in figure 5.2. This is a scenario where one source signal provides important data that every other signal depends upon. The interesting effect of this is when that specific source signal emits a new value, it will trigger an immediate high load of updates for all of its children.

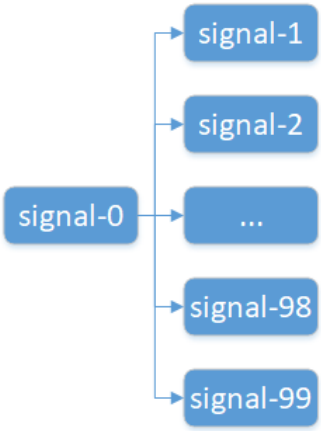

Figure 5.2: A fan-out topology: one signal with a large amount of children

5.2.3 Square

The square topology is expected to be the most common, it provides a signal graph that is not extremely deep such as the linear topology nor is it very wide like the fan-out variant. In this topology, signals have a variable amount of parents and children, shaping the graph into a square. For the purposes of our benchmarks, we will use an exact square graph for simplicity.

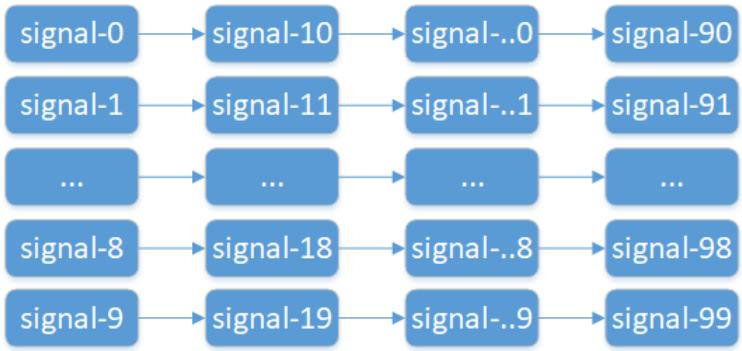

Figure 5.3: A square topology: The signal graph is equally deep as it is wide

5.3 Benchmarks

All of the benchmarks in this chapter are run in the Racket VM with unlimited memory (16 GB RAM physical limit) and on an AMD Phenom II X4 955 processor (Quad core, 3.2 GHz). Three sample programs were created with around 100 signals each, each of which implementing one of the three topologies as shown in the diagrams. These programs were each executed three times for 60 seconds, taking the averages of the output to produce the charts in this chapter. A single source signal was emulated to emit 10 million values per second and the output nodes of the signal graphs were inspected to collect information on how many values were propagate through the graph and at what rate the update mechanisms could keep up with the stream of incoming data.

5.3.1 Without dataflow engine

We will first evaluate the runtime that implements an update loop in the interpreter itself and does not use the dataflow model to keep its graphs up to date. These are the tests for our first runtime for FrDataFlow.

Latency

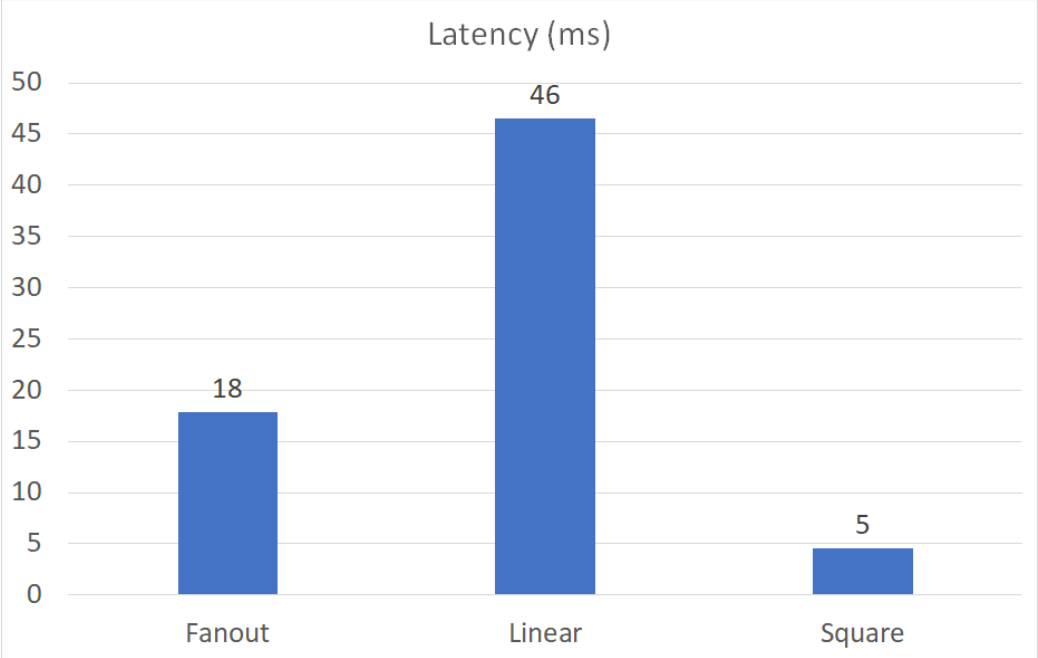

Figure 5.4: The latency of signals in FrDataFlow without the dataflow engine

Inspecting these results, we conclude that the linear graph show the highest latency, which is expectable: values in this topology travel the longest path of signals before they reach the end. On average, it took nearly 50 ms for a value to propagate from the source signal to the last output node. More interestingly, values in the fanout topology seem to take longer on average to reach the end than in the square variant. This is counterintuitive, because every value in the fanout graph only needs to pass two signal nodes before it reaches the end. Since the fanout graph contains 99 output nodes and the square graph only 10, it would seem that this gives the square graph a slight edge: these 10 output nodes have to wait less long to be given the chance to announce their output.

Throughput

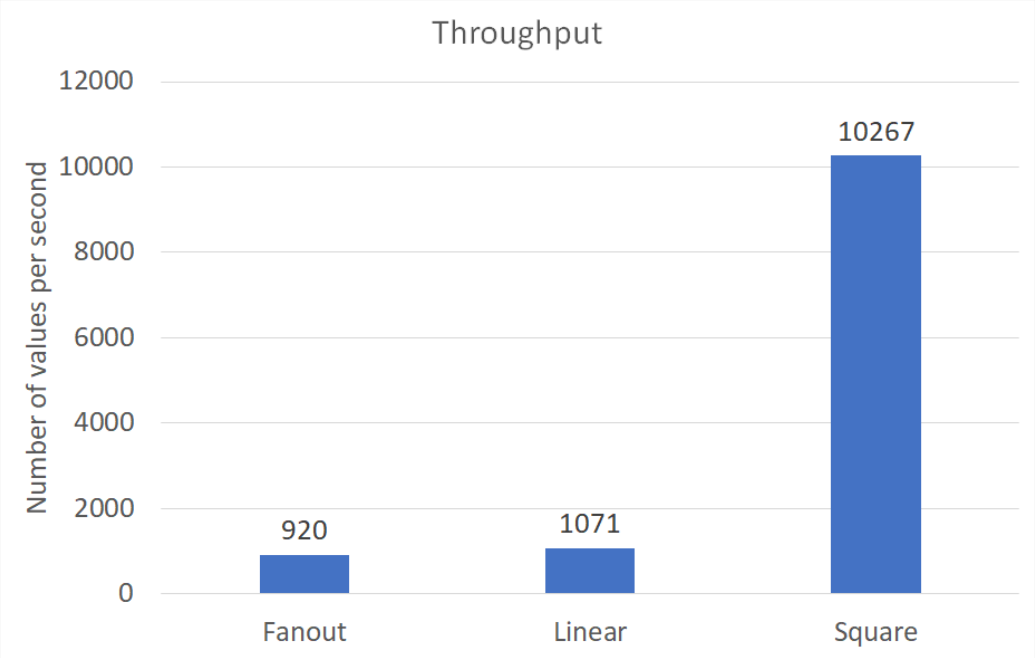

Figure 5.5: The throughput of signals in FrDataFlow without the dataflow engine

Even more so than with latency, the square topology is the clear winner when it comes to the sheer amount of values it can push through the graph per second. The square graph can push out more than 10 000 values per second and is an order of magnitude more efficient compared to the fanout or linear approach, who only managed to process 1 000 values by and large. This makes sense, because every update loop iterates over all the signal nodes once. In the square topology, this means it can propagate 10 values to the end in one loop. The linear version on the other hand can only process one, because when the iteration is happening it is really pushing forward one value from the first signal all the way to the last. Similarly, the fanout approach results in one loop pushing a single value from the first signal to all other signals. The square topology, by splitting up the graph in 10 linear parts, essentially parallelizes this work.

5.3.2 With dataflow engine, single core

In this section, we repeat the same tests of the previous section but this time using the runtime that sits atop the dataflow engine. The programs are

exactly the same, but they are now compiled to dataflow instructions and invoked there. These are the tests for our second runtime for FrDataFlow.

Latency

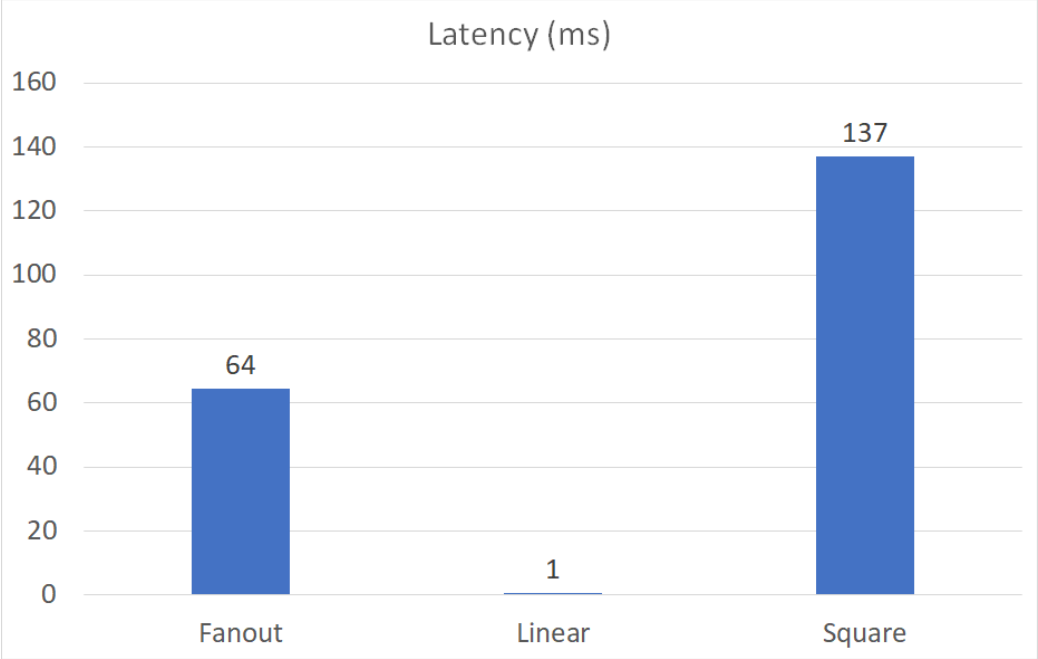

Figure 5.6: The latency of signals in FrDataFlow with the dataflow engine

The first thing that is immediately obvious is the general increase in latency, except for the linear topology. We observe an average of 50 ms extra latency because we now translate to dataflow instructions. This was to be expected: the use of a dataflow engine brings with it some extra overhead such as the management of the token queue. While 50 ms is considerable, it is not a show stopping performance penalty and is thus acceptable for the gains that can be made in other spaces. Special mention goes to the linear topology, which seems the only approach to actually decrease the average latency. The dataflow engine seems optimized for a linear cascading chain of tokens when it comes to latency.

Throughput

We observe that the general throughput using the dataflow engine is more efficient than the first runtime. Especially the fanout and linear approaches benefit here and see a tripling of their output. The square topology on the

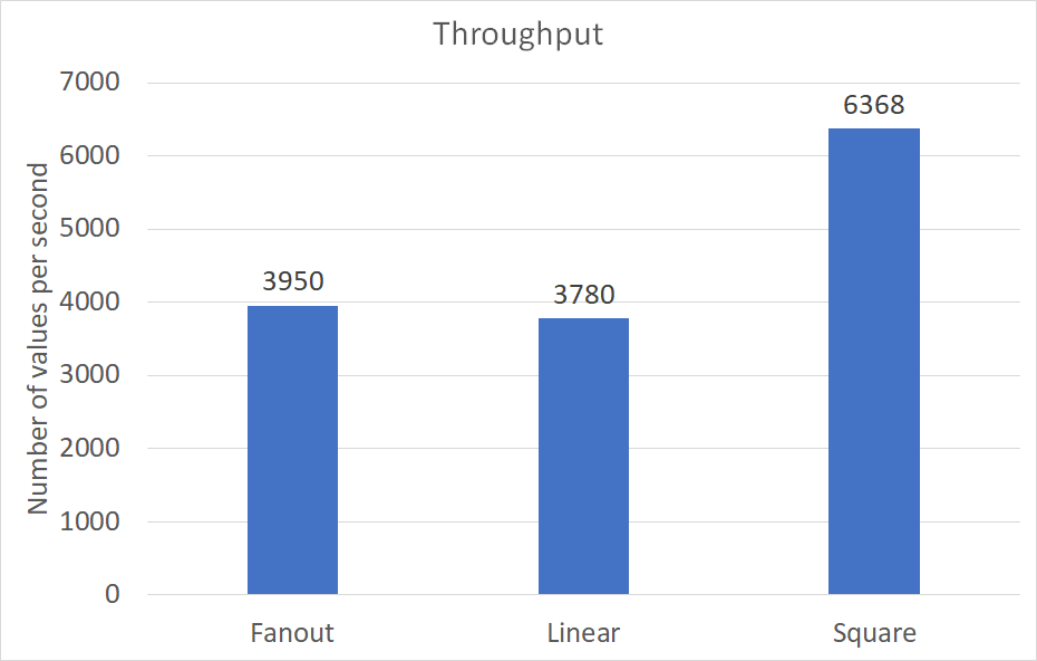

Figure 5.7: The throughput of signals in FrDataFlow with the dataflow engine

other hand is less advantaged by the second runtime and takes a small hit in throughput compared to the first runtime.

The major reason for this increase in throughput can be attributed to the compilation step of the signals that happens in the mapping algorithm. In the first runtime, the update lambdas associated with each signal are fed into the metacircular evaluator which has to lookup variables in the lexical scope and generally introduces a significant overhead just to call a lambda and pass on a value. When running atop the dataflow engine, these update lambdas are fed natively to the dataflow engine which has no knowledge of the metacircular evaluator, which allows it to skip most of the evaluation steps that are necessary in the first runtime.

The reason why the square topology does a little worse is because the update loop in the first runtime is - by accident - ideally optimized for square topologies, while the dataflow engine is not: tokens are just processed one by one, which levels the playing field for all three topologies.

5.3.3 With parallelization, four cores

In this section, we repeat the same tests a third time but this time using configuring the dataflow engine to spread its work across 4 cores. The engine does this by making use of Racket Places [11], which starts up n runtimes where n is a number lower than or equal to the number of available processor cores.

Latency

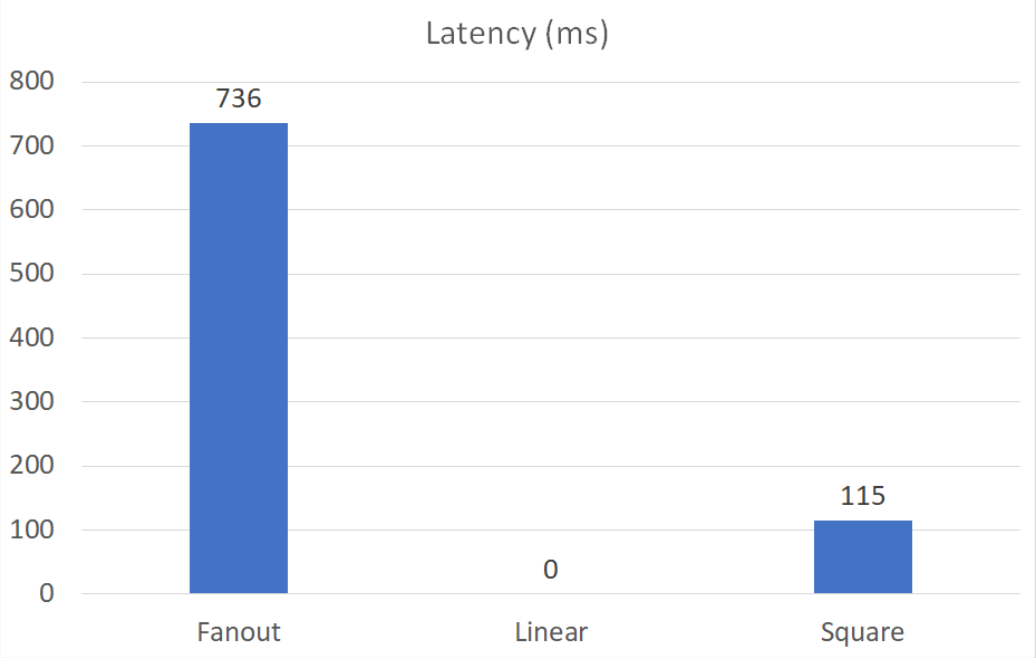

Figure 5.8: The latency of signals in FrDataFlow with the dataflow engine running on four cores

By switching to four different processor cores, the fanout topology was most negatively impacted. By distributing the queue of token across four different instances, tokens took nearly a second each to reach their destination. The linear approach on the other hand strongly benefits from the parallelisation: we see that each chain of values that travels to the end of the graph is still extremely fast. Latency wise, there was no penalty for the linear and square topology, resulting in a net loss of throughput.

Throughput

Even though the latency of the fanout approach is quite worse than its non parallel alternative, it did positively impact the throughput of this topology. Since this approach results in immediate bursts of new tokens, the fact that four processes are simultaneously picking these tokens up gives it an edge over the other topologies. Surprisingly, the square topology suffers most from the split up into multiple parallel instances of the dataflow engine. It seems that the communication between the instances causes more overhead in this topology than the benefits of distributing the work. Racket channels seem to cause a lot of delay to get values to the end of the graph.

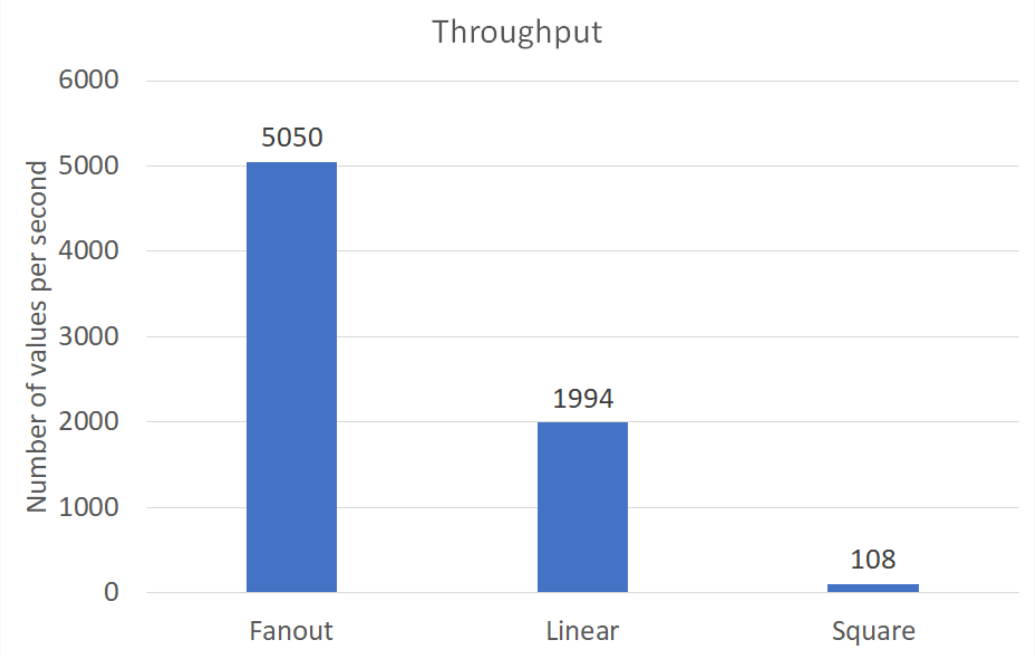

Figure 5.9: The throughput of signals in FrDataFlow with the dataflow engine running on four cores

5.3.4 Discussion of results

From the results of our benchmarks, we can first conclude that latency generally increases when running atop the dataflow engine, especially when running it in parallel is added to the equation. This was to be expected: the dataflow execution model goes through more phases and needs more bookkeeping than the update loop of the first runtime to support the same behavior. On the other hand, it provides more scalability. We see that the

throughput globally increases when running atop the dataflow engine, although the introduction of parallelism does not result in better throughput. We attribute this to two characteristics of our practical implementation:

1. Racket places communicate across threads using channels, which does not come cheap. This communication overhead, with certain topologies, unfortunately is more detrimental for the throughput than the benefits of running the engine in parallel.

2. In our benchmarks, the signal update callbacks merely pass on incoming values rather than doing a meaningful computation. While this is common in reactive programming, it does mean that calling these callbacks is extremely fast and that parallelizing them will only be beneficial if the overhead of parallelisation is negligible.

5.4 Conclusion

In this chapter we presented numbers that show the performance of three sample programs that implement different topologies. We tested these programs using a simple update loop in the first runtime, using the dataflow engine on a single processor and using the dataflow engine on four processor cores. Collecting these results, we conclude that while the signs of scalability are there, in our current setup the benefits are not visible. As long as the overhead of running in parallel is not reduced to the bare minimum, it will require large signal graphs and a large amount of parallellism to be more efficient than simple single threaded approaches in terms of latency and throughput. In our implementation, this overhead is still too considerable to make the effort worthwhile.

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 signal graphs that can be made in FrDataFlow are still static, e.g. 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. Imagine a button which starts a timer. When this button is pressed, a timer should start running and continuously update the screen with the elapsed time.

This button can be represented as a signal of clicks. Every click is essentially an event, which in terms of reactive programming can be represented as an event being emitted by a signal.

A timer is also a perfect fit for a signal: it is a signal that continuously emits the current elapsed seconds. The tricky part however is that we only want to create this timer when the button is pressed. That means we want to lift the button signal and create a new timer signal only when it is pressed. Maybe another button click stops the timer, or starts a second one, but that is outside the scope of this example.

This pattern of dynamically creating signals is quite common in reactive programming, and unfortunately impossible in FrDataFlow today. A possible improvement would be to enable 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. The requirement for a static graph is currently one of the limitations of the dataflow model. There is currently not a single dataflow engine who supports this. It would technically be possible, although it would introduce a considerable amount of bookkeeping overhead. For values that are already being processed by the dataflow engine at the time this reconfiguration happens, it would highly depend on where in the update graph these values currently are to determine if the reconfiguration will be in time to apply for these values. It is therefore impossible to guarantee the correctness of the values already in the update graph at the time of reconfiguration.

6.3 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 6.1.

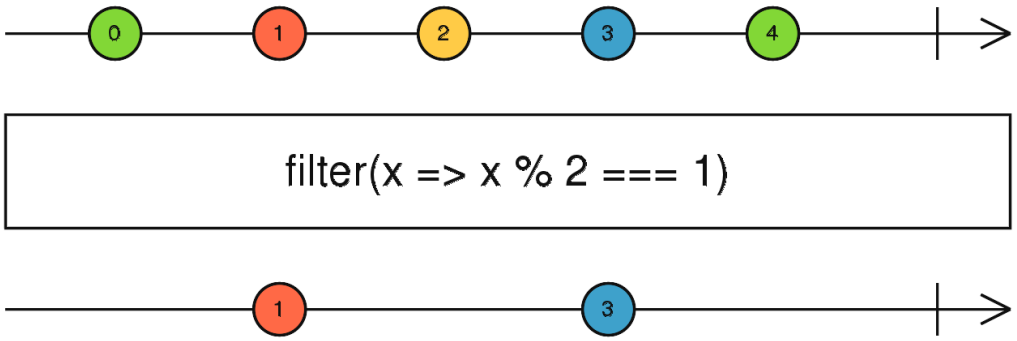

Figure 6.1: A timeline diagram of the *filter* operator in RxJs

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. Unfortunately, the current design of FrDataFlow does not allow the production of zero or more than 1 output token for 1 input token, because of the design of the lift operator. This operator would have to be redesigned to use a sort of sink model where values can be pushed into, allowing each lift operator to decide the amount of output values it produces per incoming value. The value of each signal would then no longer be determined by the return value of the lift operator, but rather by the values that get pushed into the sink object that is provided to the lift function. This approach has not been validated yet, but we propose

that the model does support it.

6.4 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. The dynamic creation of signals is possible in theory, but would have to make trade-offs regarding the correctness of values already being processed by the dataflow engine. Secondly, filtering could be implemented using a sink model in FrDataFlow, with the necessary updates to the mapping layer so it would generate tokens based on the values that get pushed into the sink. However, both of these proposed solutions have not been explored and are therefore not validated.

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. Another source of inspiration was FrTime, a DrScheme implementation of reactive patterns with some subtle yet powerful differences. Its focus lies more on the implicit transformation of regular programs to reactive programs, while FrDataFlow remains very explicit about the existence of signals.

Finally, we discuss the virtual machine design of the dataflow engine that FrDataFlow uses to execute its instructions. This VM builds on the principles of tagged token dataflow and provides a platform upon which the reactive nodes in FrDataFlow are scheduled and executed.

7.2 Reactive programming

7.2.1 Elm

In 2012, Evan Czaplicki wrote his dissertation on Elm [5], 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.

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 are 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:

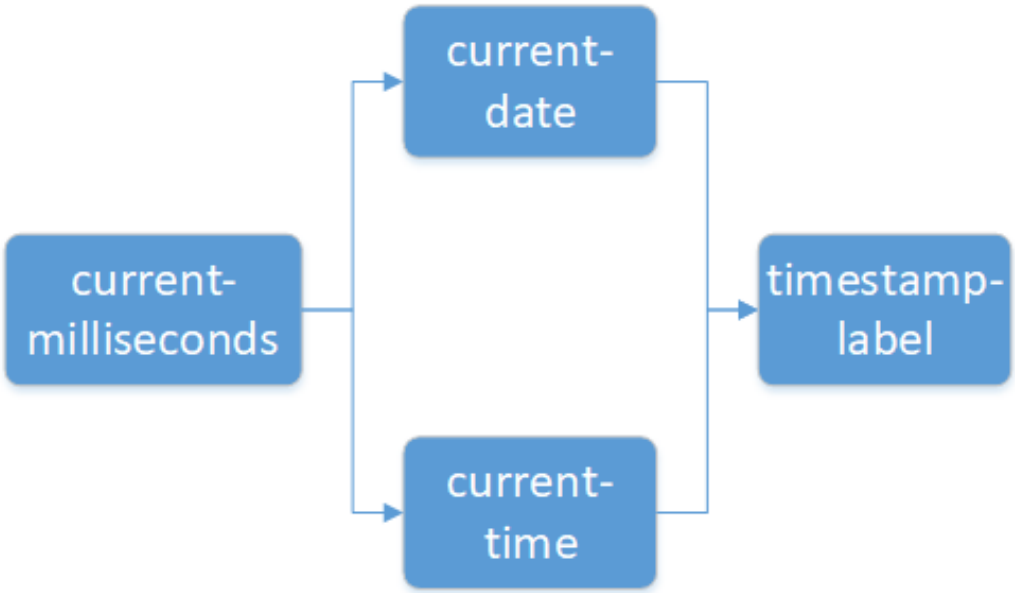

Figure 7.1: The current timestamp is based on two different computations of *current-milliseconds*

- 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

[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.

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) 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 worthwhile though, because of the amount of overhead to orchestrate this and the possible security ramifications.

7.2.2 FrTime

Another project that is closely related to FrDataFlow is *FrTime*. This is a reactive programming language that implicitly implements signals for primitive operators. The idea is that applications written in regular Scheme could be switched over to FrTime to introduce reactive programming without having to rewrite the code. Contrary to FrDataFlow, expressions in FrTime do not require explicit use of the lift operator, but are implicitly used when any of the primitive operators are used. For full effect, FrTime provides a REPL (Read Eval Print Loop) console which takes expressions and immediately prints the result. There is an extra feature though: values which are signals keep getting updated in the history. When for example the plus operator is applied to a fixed number and the current seconds, this creates a signal which continuously updates with the result, inside the console. Rather than evaluating the expression and returning a signal like in FrDataFlow, FrTime immediately returns a value and keeps updating that value, even inside the REPL history.

```
> seconds
1496651429
> (even? seconds)
#f
> (+ 10 seconds)
1496651439
```

Listing 7.1: REPL in FrTime

See the listings 7.1 and 7.2. In the first sample, some commands are entered into the REPL and their output is shown below each command. Note that the listing 7.2 simply shows exactly the same screen from listing

7.1 but 5 seconds later. It is not revealed to the user that the expressions he is using are built with signals that update in the background. It only becomes clear, as time goes by, that these values automatically update in the REPL.

```
> seconds
1496651434
> (even? seconds)
#t
> (+ 10 seconds)
1496651444
```

Listing 7.2: REPL in FrTime, 5 seconds later

Push-driven evaluation

A major difference between FrTime and FrDataFlow is its update model. FrDataFlow implements what is called a push driven evaluation. This means that parent signals - or producers, as FrTime calls them - have references to their children (dependents in FrTime). When a parent signal gets updated, it can follow along the edges of its dependencies to ripple the change forward in the graph. Since child signals do not hold references to their parents, closures are used during the signal construction process to capture references to these parents in the callback procedure.

In FrDataFlow, callbacks and closures are not an essential part of the update mechanism. All signals are stored in a topologically sorted graph, where each signal can access its parents and children. When a new value propagates through the graph, the update loop simply grabs the necessary information from the parent signals to provide to the child.

FrTime's callback model in combination wi]th their queue based update algorithm actually comes with a string of performance problems, which they try to address by *lowering* [3] their operators, essentially collapsing multiple operations into one to reduce overhead and to minimize the graph size. Although this considerably improves execution time, it unfortunately also reduces parallelization opportunities by growing the size of one signal node. In FrDataFlow, no such optimizations are considered.

7.3 Dataflow Model

7.3.1 Virtual machine design for the execution of languages on dataflow machines

In [10], a virtual machine design is presented to execute instructions on dataflow machines. This VM tries to offer a framework in which the principles of tagged token dataflow are strictly adhered to, while allowing full access to the tokens and execution context from within a single dataflow node, i.e. an operation. This allows the creation of operations which can manipulate the destination they send data to, enabling conditional statements and dynamic flow in general. To enable closures, operations also receive access to the active working memory of the current context. In short, this design is intended to support all common flows expected in a high level language, such as function calls, closures, recursion, exception handling, etc.

For FrDataFlow, a minimal version of the engine described has been implemented in Racket, to avoid the overhead of having to communicate between Racket and Python.

7.4 Conclusion

In this chapter, we presented other research that closely relates to and strongly inspired FrDataFlow. Elm, the web programming language with a strong focus on FRP, introduced many of the same concepts also applied in FrDataFlow. FrTime has similar semantics, but chose to hide its internals more from the end user. For the execution of our signal nodes, we applied the virtual machine design of DVM in Racket to map our reactive nodes as dataflow operations.

8 Conclusion

In our research for this dissertation, we explored the initial steps needed to map from reactive programs to the dataflow execution model. These steps included creating our own experimental language FrDataFlow, writing a mapping algorithm from a reactive graph to a dataflow graph and executing the latter atop a data flow engine as described in [10]. This chapter will revisit the original problems we set out to tackle, the solutions we proposed and conclude with some closing remarks.

8.1 Revisiting the problem statement

Reactive programming is an event-first programming paradigm well suited for reactive systems (e.g. user interfaces, robotics, etc.) that provides the concept of signals which can be composed using declarative operators. These signals are a common abstraction over I/0, events and asynchronous computations and provide a single interface to model these concepts. Efficiency and timing is essential to create responsive and reliable reactive systems. However, the majority of reactive programming implementations are not designed to support parallelism, which is necessary to maximally utilise the processing power of modern processors. We propose to run reactive programs atop the dataflow execution model to make maximal use of the parallelization abilities provided by the underlying host. This execution model invokes instructions

when the inputs are available and does not provide any form of global state, allowing instructions that do not share data dependencies to be invoked side by side.

8.2 Contributions

We implemented an experimental Racket based language called FrDataFlow that adds reactive concepts to a subset of Racket constructs. This language was implemented using two different interpreters: one which directly implements the reactive mechanism in Racket and serves as a reactive runtime by manually keeping the graph of signals up to date using an update loop, as described in chapter 3 - Language. The second implementation delegates the responsibility of keeping the reactive graph up to data to a dataflow engine implementation by mapping the signals to dataflow instructions in such a way that the signals behave the same as in a regular reactive system.

More specifically, we provided the following contributions:

An interpreter for a reactive language
: We designed and implemented an interpreter based on FrTime, but which chooses to provide explicit reactive operators rather than implicitly applying reactivity. This interpreter applies a traditional evaluation model to keep signals up to date, in the form of a signal update graph.

A mapping algorithm from signals to dataflow instructions
: We implemented an algorithm that maps the concepts of reactive programming to instructions in the dataflow execution model and solves a major mismatch between the two: the consumption of arguments in dataflow, as described in Chapter 4. Furthermore, we showed that reactive programs can be correctly translated to dataflow instructions so that they behave exactly the same.

Comparison of a traditional reactive evaluation model and signals atop the dataflow execution model
: We evaluated the performance of both approaches in terms of latency, throughput and scalability and showed that the platform overhead of parallelism, for now, does not outweigh the benefits of parallelism. We do note however that single threaded dataflow execution already brings with it benefits in terms of throughput, although at the cost of latency.

8.3 Final remarks

The ultimate goal of this dissertation was to explore the possibilities of executing reactive programs atop a dataflow engine, in the hopes of achieving highly parallel execution of these programs.

We notice that high level paradigms such as reactive programming are becoming increasingly popular, but they usually come with a performance cost due to these abstractions. In reaction to this, we tried to address these concerns by proving that the abstractions reactive programming makes can be correctly translated to a horizontally scalable execution model, namely a dataflow engine. In doing so, we touched on an unexplored subject in this space, making a meaningful contribution to both the world of reactive programming and the dataflow execution model.

We look forward to seeing the further evolution of both in the future and the new insights they bring to the world of software development.

A

Your Appendix

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill Book Company, London, UK, UK, second edition, 1999.

[2] K. Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39(3):300–318, March 1990.

[3] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A Static Optimization Technique for Transparent Functional Reactivity. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, pages 71–80, New York, NY, USA, 2007. ACM.

[4] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems*, pages 294–308. Springer, Berlin, Heidelberg, March 2006.

[5] Evan Czaplicki. Elm: Concurrent FRP for functional GUIs, 2012.

[6] Joscha Drechsler and Guido Salvaneschi. Optimizing Distributed REScala. 2014.

[7] D. Harel and A. Pnueli. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*, pages 477–498. Springer, Berlin, Heidelberg, 1985.

[8] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in Dataflow Programming Languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[9] John Peterson, Valery Trifonov, and Andrei Serjantov. Parallel Functional Reactive Programming. In *Practical Aspects of Declarative Languages*, pages 16–31. Springer, Berlin, Heidelberg, January 2000.

[10] Mathijs Saey, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter. An Extensible Virtual Machine Design for the Execution of High-level Languages on Tagged-token Dataflow Machines. 2017.

[11] Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda pdinda@northwestern.edu. Places: Adding Message-passing Parallelism to Racket. In *Proceedings of the 7th Symposium on Dynamic Languages*, DLS '11, pages 85–96, New York, NY, USA, 2011. ACM.

[12] Arthur H. Veen. Dataflow Machine Architecture. *ACM Comput. Surv.*, 18(4):365–396, December 1986.