A domain-specific dialect for financial-economic calculations using reactive programming

Master's thesis in Software Engineering
by
Christian Marheim

Department of Informatics at
University of Bergen, Norway

Western Norway University
of Applied Sciences, Bergen, Norway

June 2017

Abstract

This thesis investigates to what extent reactive programming is suited for the implementation of systems that support banks in providing financial-economic advice to their private customers.

To this end, we built a small such system in Hotdrink, as a proof-of-concept. This small system is a simplification of a much larger system that has been developed by Delfi Data, and which is overdue to be rewritten from scratch. Our system was evaluated on the commercial criteria readability, maintainability, extensibility, and efficiency.

It turned out that readability and maintainability were quite high, even allowing non-developers to understand most of the code. However, extensibility is limited for financial calculations in general and especially for parallelization and encapsulation. With regard to commercialization, the solution was too immature to be evaluated on efficiency. Our conclusion is that reactive programming alone is too restrictive for solving financial-economic problems.

Notwithstanding the above, as a by-product we developed a prototype which source-to-source compiles our new high-level domain-specific JavaScript dialect to reactive calculations.

Acknowledgements

I would like to express my deepest gratitude to all the people who have helped and otherwise supported me during my work.

Especially thanks to the excellent professors Marc Bezem and Jaakko Järvi who have guided me through the process, with their in-depth understanding of computer science and decades of useful experience.

I also want to name a few of my colleagues for their help and wise opinions. This thesis is the product of commercial software development and they have been critical to its success. I want to thank my department manager, Trude Stadheim, for all of her and Delfi Datas support.

They have allowed me to complete my work with an acceptable quality in a period where I have been struggling with motivation after 8 years of combining school and work. I want to thank Svein Ove Båtnes for his time and godlike financial competence. I wish him the best of luck with his future JavaScript projects.

I also want to thank the group of highly proficient architects at Delfi Data, Øystein, Nils, Bjarte and Audun. Special thanks to Martin Larsson for discussing these topics with me and sharing his experiences from solving these exact problems multiple times before in Rådserver. These discussions have been invaluable for my understanding of the domain and identifying achievable features and dead tracks.

I also want to thank my family, Rune, Kari, Andreas, Markus Theodor and Caisa for their support and understanding.

Last thanks go out to my closest friends Espen, Shake, Oen og Tony for encouraging me and respecting my need for study time ☺

Contents

List of figures

1 Introduction

1.1 Chapter organization

Chapter 1 introduces the problem statement and motivation. We will briefly explain some concepts in the financial-economic domain and introduce the term “reactive programming”.

Chapter 2 introduces Hotdrink as a library that enables reactive programming. We will look at some simple examples and their limitations.

Chapter 3 introduces higher level of abstraction by creating a new domain-specific dialect of JavaScript. The new DSL is compiled to Hotdrink API calls. We can now solve more complex examples using our DSL.

Chapter 4 uses the tools and DSL from chapter three to solve a more complex real life case. We discuss the implementation of a time dimension and some of its difficulties.

Chapter 5 evaluates the solution from three perspectives: the commercial, the academic and the main research problem. We will also take a look further, outside of this thesis’ scope, and discuss how reactive programming can improve the whole software development process.

Chapter 6 and 7 are Conclusion and References.

1.2 Problem statement and motivation

Financial calculations are quite simple. Mathematics is rarely the limiting factor. The problems in financial calculations arise from the combination of large datasets that consists of both rules and input data. When should calculations change? Which parts are affected by the new rates? Will the input data in production still be valid after adding new business rules? How do we know the results are still correct? In work with financial calculations, focus must be in enabling change.

We want to use a new technique that lets us calculate correct values in the correct order. This way we can avoid many problems in imperative programming such as managing state, synchronizing, multithreading and other technicalities.

My employer, Delfi Data, have already tried solving these problems using techniques like imperative programming, transaction based calculations and static data structures like arrays. These techniques do work but they are hard to understand, hard to create and even harder to maintain. They also have limited extensibility but they do allow for certain optimizations.

In this thesis, we explore the possibilities and difficulties of solving these problems using the reactive programming paradigm [1, pp. 1-2]. By using declarative programming together with reactive, we should be able to focus on entities and their dependencies. Hopefully this allows us to create a system in terms of financial terminology and modelling with high cohesion instead of computer science technicalities like data types, concurrency, error

handling and optimizations. The latter usually produces extreme amounts of code in comparison.

This should greatly ease the cost of development, verification and maintenance. The big question is, however, at what cost? What technical limitations will arise from this reactive approach? How will all the strange corner cases and artificial concepts in finances fit and be implemented in such a system?

Will the limitations be acceptable? Will they appear acceptable but actually be critical obstacles later in the project? Are there any opportunities regarding optimization and extension?

All these questions have already been answered by the current legacy implementation, a component called “Rådserver”. It directly translates to “Advice-server” from Norwegian and it basically does all the computing. It is written in Borland Pascal and is a Windows 32-bit only application using older structures like DLLs, OLE and COM objects. It does not suffer from problems like “DLL Hell” or having a massive code base but being so old complicates some critical issues.

The “Advice-server” only runs on older Windows systems which are no longer supported by Microsoft, a big no-no for the banking sector. To make matters worse, the old tools used to develop the application are so outdated they are only able to compile using Windows XP or older. Support for the latest version, Windows XP, was completely stopped in 2014.

This legacy component “Advice-server” can compute everything we need. Since we could not afford to make a new one we integrated the old “Advice-server” with our new solution. We already had a dire need to replace the “Advice-server” but now it is even more costly because it is used in the new solution too. This was a strong motivation to begin a new implementation from scratch because we knew our current path was unsustainable.

We had already tried most other technologies and reactive seems to be popular and matching our domain. All stakeholders agreed on a “proof of concept”-approach and the following become my main research question:

Give a proof of concept for financial calculations using reactive programming. It will be evaluated on readability, maintainability, extensibility and efficiency of execution.

1.3 Goals and research questions

Which types of financial calculations are appropriate to be solved by this technique? Can all problems be solved? Can problems be solved partially in the reactive framework? What problems can we expect to arise from such techniques in the long run? Most importantly, how will the reactive components enable the implementation of time?

Time is a difficult concept to implement in a reactive context because it demands multiple or dynamic timescales. This requires every data element in reactive system to have one or more connections to other nodes. In other words, we need to dynamically handle multiple many-to-many relationships.

Sometimes the relationships are day-to-day accuracy, other times weekly, monthly or yearly. This would not be very difficult if we only had to worry about time but we also need to fit this dynamic concept into the static hierarchies of accounting and aggregation. A simple visualization of this domain could look like following figure:

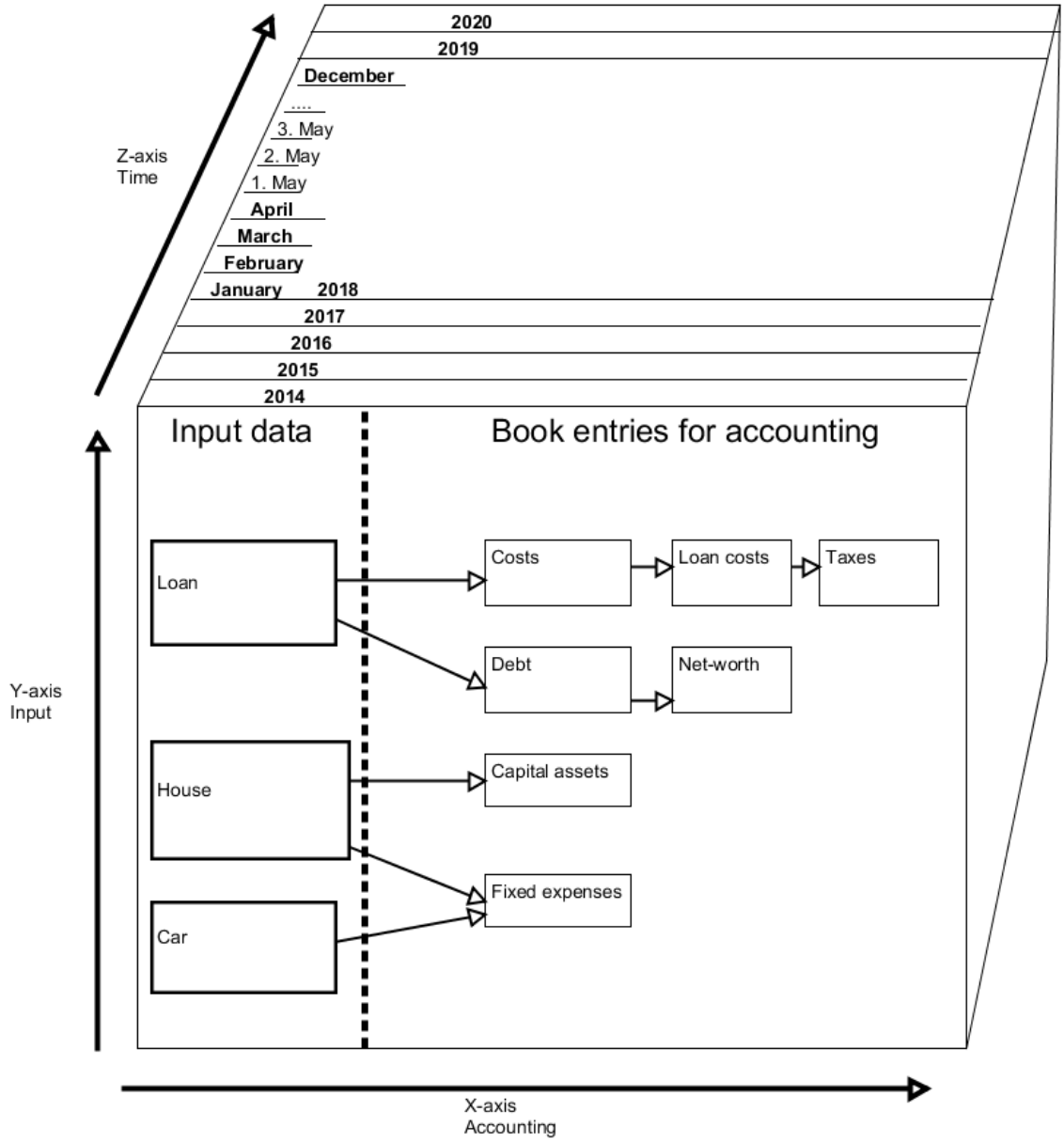


Figure 1: Simplified dependencies

In the figure above we see a structure where input data is added to the accounting as book entries. This is simple enough alone but once we add time it becomes a lot more complex.

For example, we need to make sure that changes for a loan updates the yearly taxes and monthly payment costs, but a capitalization on the 5th should not be included if it is before the mortgage payment date. Capitalization on the 5th means the bank calculates and add the interest to the balance on the fifth day in a month, but not necessarily ask the customer to pay it. This is a common problem in finances. We will revisit this figure and elaborate on its concepts in next subchapter.

Researching how we could implement a dynamic timescale was and still is the most important. However, as I had to start completely from scratch, creating a working solution was unlikely. The more likely result is experience and insight into the reactive domain. This is expected to give indicators on whether reactive programming is useful for our core domain problems or not. We all agreed that the best goal to achieve this insight was to create a "proof of concept"-application.

1.4 Evaluating a family economy

A family economy comes in many shapes and contexts. Some families need help to lower costs so they can buy a house. Others need advice on how to invest their fortune. Some family economies contain one member, others more than 10. Some families know how much they spend each month, some only know how much is left right now or not even that! In summary, we are trying to have a general method of evaluation where both the number of parameters and the accuracy of them are unknown.

To deal with this uncertainty we start with what is known, how much does it cost for an average person to live and own a house or a car. People mostly spend about the same amount to feed five people or keep a given car maintained. This means we can calculate what the average monthly costs are for a given family composition. Then we can adjust the pregenerated analysis according to the deviation for that particular family. The following is an example screendump of costs with standard rates, and incomes like salary that cannot have standard rates:

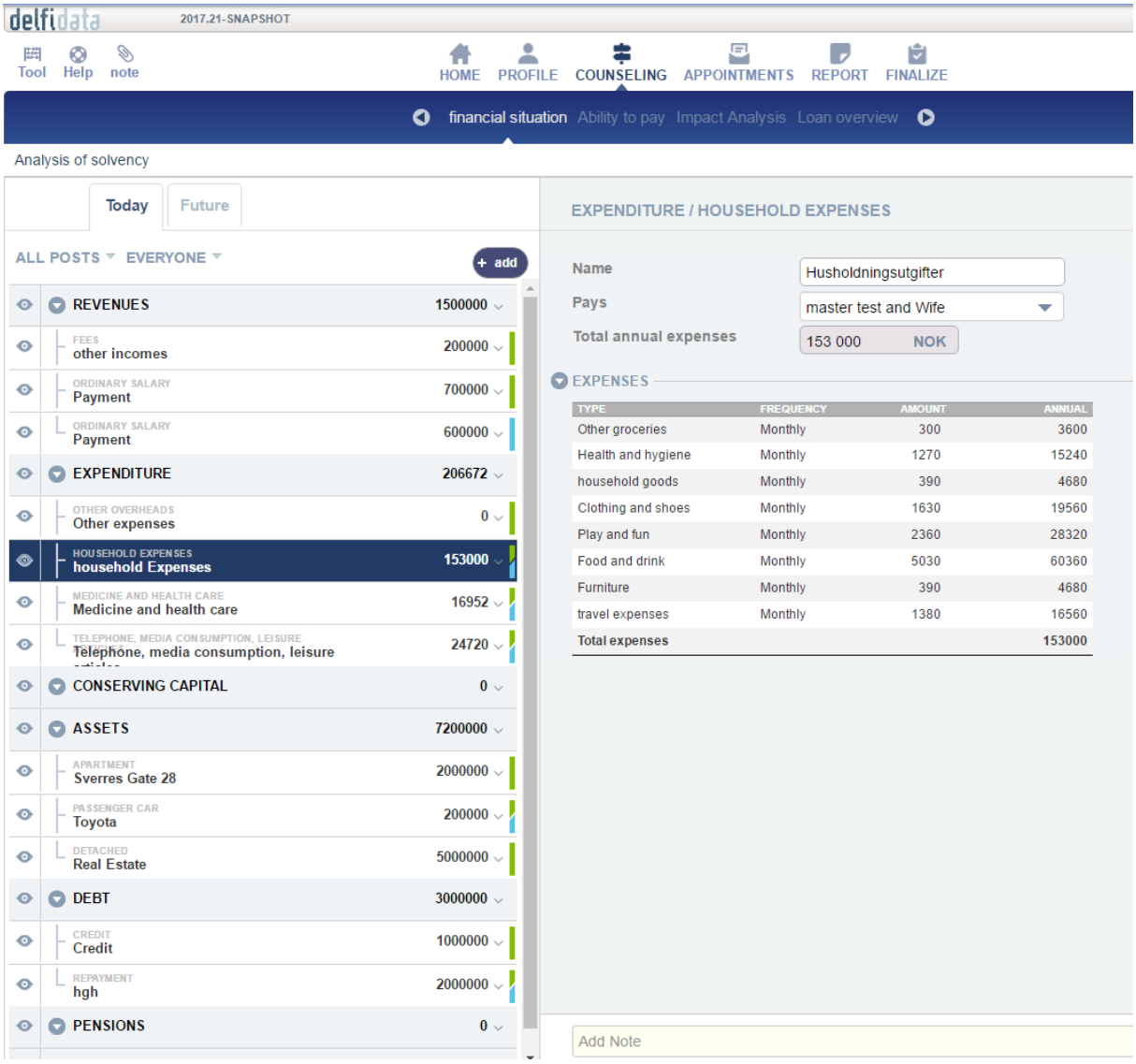


Figure 2: Screendump from current solution

As we can see from the image, many expenses are very accurate but major factors like salaries and assets are not. They have to be entered manually but costs like medicine can use standard rates.

The techniques for this evaluation have been informal because of lacking legislation in documentation and accuracy. This changed with the EU's "Markets in Financial Instruments Directive" (MiFID) directive in 2004. It was later extended in 2006. Now it is illegal to give financial advice to private customers without giving detailed documentation and evaluating the customers understanding of complex financial products. The demand for standardized software increased and grew into the software which is the source and result of this thesis.

Many of the financial rules we are describing throughout the thesis are general and can be applied to any sort of economy. However, this thesis is only trying to solve practical problems within the domain "family economy". We will therefore ignore other more general aspects like business accounting, return on investment, cash-flow analysis, etc. This is a result of the "proof of concept"-scope.

A potential full implementation however, should model reality as close as possible to ensure very few limitations. The system is expected to perform just as well within the small and medium-sized business domain where cash flow is extremely important. A cash flow analysis has no value in family economy because monthly salary is the only income for most families.

We will now look at some concrete problems we could be trying to solve. Four very regular cases:

> *How much can I loan?*
> *How can I invest my income and fortune?*
> *How can I invest my fortune during pension?*
> *How is my economy affected by changes in my family situation? Examples can be a new child, a marriage, a divorce, death or disability.*

Many of these questions are simple but data-heavy simulations where you work through some data structure resembling a spreadsheet. An example with a three-dimensional matrix:

- **1st dimension - Input data**
  - How much do I make?
  - What savings account do I have and how are they growing?
  - What is my house and car worth?
  - Who lives in your family, how old are they and what gender?
  - Extraordinary incomes or costs?
- **2nd dimension - Balance sheet/accounting structure**
  Like in accounting, all input is aggregated into accounts:
  - Sum taxes from all income and expenses
  - How much increase in interest rate can I handle?
  - Monthly accuracy for liquidity purposes, yearly for investments
  - How much do I own? (net worth)
  - Which parts of the income are my wife's?
  - Which parts of the expenses are my wife's?
  - How does insurance against death or loss of income affect us?
- **3rd dimension - Time**
  - Yearly values are always relevant because summaries are at least annually
  - Liquid assets on a daily level of detail is very relevant for businesses because it indicates how close it is to bankruptcy
  - Liquidity can matter on a daily level when moving while having 2 loans
  - Salary and a lot of costs follow a monthly frequency although they can occur on different days within a month
  - Certain scenarios like moving or death have big implications for most of the calculations
  - To calculate interests interest and efficient interest rate for a legally binding payment plan we need to have daily accuracy in loan calculations

None of these questions point us in any direction on data structures like node lists, arrays or trees. We only know for sure that they are possible to answer with a spreadsheet because

that is how it is done today. If we visualize the three-dimensional structure above, we will have a simplified version of the current “Rådserver” solution(see page 7 chapter 1.2 for details):

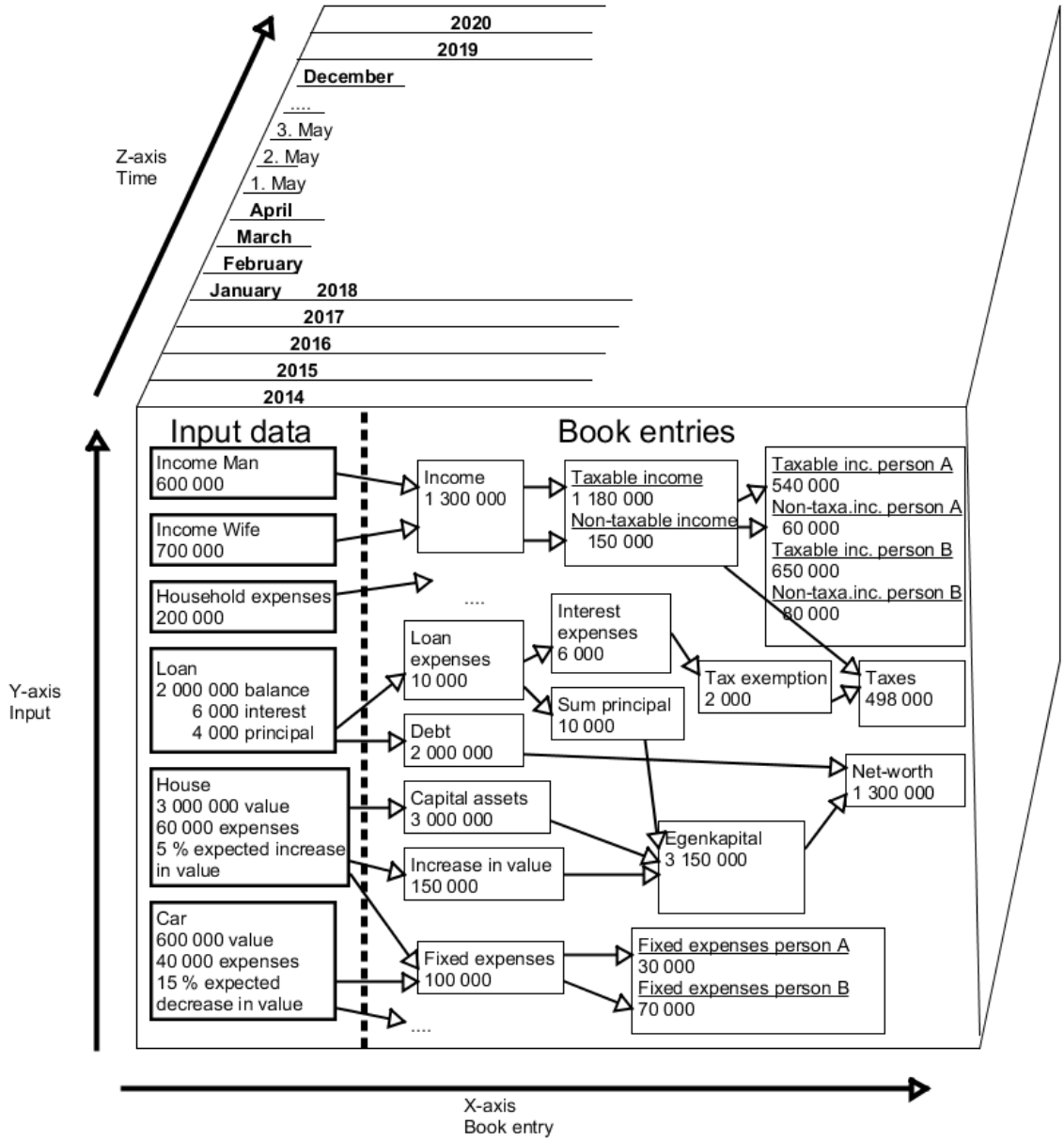


Figure 3: More complex dependencies

Figure 3 is a more complex but still simplified version of Figure 1: Simplified dependencies. The structure went from three inputs and eight dependencies to at least 14 inputs and 30+ dependencies. There are a lot more details, more levels of dependencies, differentiation between persons and more. Introduction of more concepts have a tendency to introduce more permutations of data, use cases and more.

The listed examples on previous page show perspectives and their requirements for the calculations. The requirements can be overlapping, have small variations and even be self-

contradicting. There are typically many business rules in every result and they require updates according to changes in legislation or rates.

This makes up for cumbersome processes in change management, development and maintenance. It always involves a developer in all stages and usually product owners too.

1.5 Development scoped to maintainability and extensibility

Many apps and small programs have a short lifespan. The opposite is true for most enterprise applications, they have long and sometimes lively lifespans. For these huge applications with multiple new features every year, the long-term factors are the most important by far [2]. Such factors include but not limited to:

> ***Architecture*** *– How does the architecture ease tasks instead of creating them*
>
> ***Development costs*** *– Cost of developers, cost of mistakes in development, cost of adding new developers, cost of deploying to production, etc*
>
> ***Maintainability*** *– How easy it is to correct, test and change the application*
>
> ***Extensibility*** *– How easy is it to extend the application, adding functional and non-functional features*

When these factors are below the appropriate level we accumulate technical debt. This is usually not apparent until later stages in the lifecycle when there is too much technical debt to ignore it. We have then reached a point where technical debt is costly enough to tell the difference between a naturally difficult task and a task that is hindered by this debt, e.g. the wrong architecture.

There are other attributes too like scalability, security, portability, usability and more but we will ignore them in this thesis. Even though we ignore them here they can be the deciding factor in many enterprise scenarios.

Because architecture and the other factors are extensive topics in themselves, this thesis will focus on the code and the functionality it can produce. Analysing a household's economy is a domain problem that is nothing like running Googles search engine, creating operating systems or libraries that will be used by thousands of developers.

Calculating a household's economy is quite simple in its nature because it can only include parameters that are relevant and measurable. Families do not have income from microtransactions. Families barely need to manage their taxes at all. Analysing a family's economy is never required in real-time.

Average families have 10 to 20 parameters that affect them noticeably. A mortgage loan is at most calculated twice a month. A monthly mortgage requires only 1000 simple calculations for a 40-year loan, piece of cake for a modern processor. The most important criteria for analysing a family's economy is up-to-date calculations that are always calculated correctly in a reasonable amount of time.

The hypothesis for the thesis is that a declarative language describing calculations done by a reactive system will handle all these issues for us. We already know that reactive systems are excellent at avoiding unnecessary calculations. We have yet to discover how fast it is and how well it handles other requirements.

The declarative language works within the constraints of the reactive system. We should then get a good indication of how those constraints fit the financial domain.

1.6 Reactive programming

Reactive programming is a programming paradigm which handles data flow well. The flow is dependent on a graph with nodes and directed edges. The nodes contain the data and the directed edges are functions which represent a dependency. A node has dependencies to all nodes that can lead to it, including transitive relations. This relation between multiple nodes and a result node can be described as a list of input nodes going through a function before ending in a single output node.

This simple relationship means that we can use declarative language to simply state our rules. The reactive framework will then solve our rules or deem it unsolvable. All issues with synchronization, optimization and calculations are handled by the framework. Because the rules can be set up in advance, being unsolvable is more of a compile problem than a runtime problem.

In a more theoretic formulation, a reactive system can be expressed as a system of equations. There is no sequence, order or state in the calculations, only implicit mathematical rules which are to be correct at all times. Let us take an example:

$$A = B + C$$

In imperative programming this would result in the addition of B and C **before** the result is assigned to A. In a system of equations or mathematics in general this would be one single change although we as human would do it in steps. This applies when using the system, usually through an API.

Internally the reactive system is executed on a computer and thus performs operations like one, step by step. This is a part where reactive systems differ in their implementations, some favour attributes like correctness over speed and response. This will be further discussed in the final chapters.

Most mathematical calculations in economics are simple. The complex and hard part is the uncertainty and making decisions based on those calculations. Because the decision making is done by a licensed financial advisor, the tools should support the advisor in his or her work.

To provide support in both simpler and more complex cases we try to make the business rules as convenient as possible by hiding the huge amount of data and extract the interesting parts. We usually skip most of the details but in certain cases they are important and have to be supported. This is a great challenge for the testers and product owners because all parts of the application must be consistent with each other.

As an example, we consider adding functionality for the asset boat. It will only require minor adjustments to be included in calculations, but a full implementation to

display the new feature in the generated reports. The boat's attributes must be correctly accounted for in both increased net worth but also expenses. It might also require impairment, the cost of deterioration in assets, which can be complex depending on type and year of boat.

It can be quite complicated when we add all these detailed business rules to an already existing application with many thousand rules. We are then required to make sure the implementation did not break anything. But it is hard to verify the absence of new side effects. It is even harder when the permutations in input data are almost infinite. We can certainly see that in production where the application processes old and often inconsistent data from the bank's core systems.

Reactive programming and its use of declarative code seems to be an excellent tool to combat the challenge of cumbersome business rules. Because the calculations are simple arithmetic operations applied on a lot of elements, they can be expressed in simple formulas. Our hope is that adding a few custom functions will solve the remaining corner cases. We then enter all the business rules into a reactive API like Hotdrink's and our problems should be solved.

2 Hotdrink

2.1 Hotdrink – a reactive framework

Hotdrink is a reactive framework written for JavaScript using TypeScript (a superset of JavaScript). It is works by creating a directed acyclic graph(DAG) where the nodes are variables and the edges are functions. The sink nodes will then contain the resulting value from its source nodes. A simplified figure for A = f (B + C):

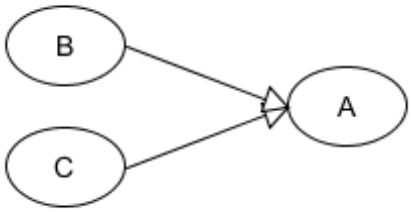


Figure 4: Oversimplified nodes

Hotdrink can express any arbitrary functional dependency between nodes. This requires the internal graph to be a bit more complicated. An example with 2 equations and multiple values would look like this:

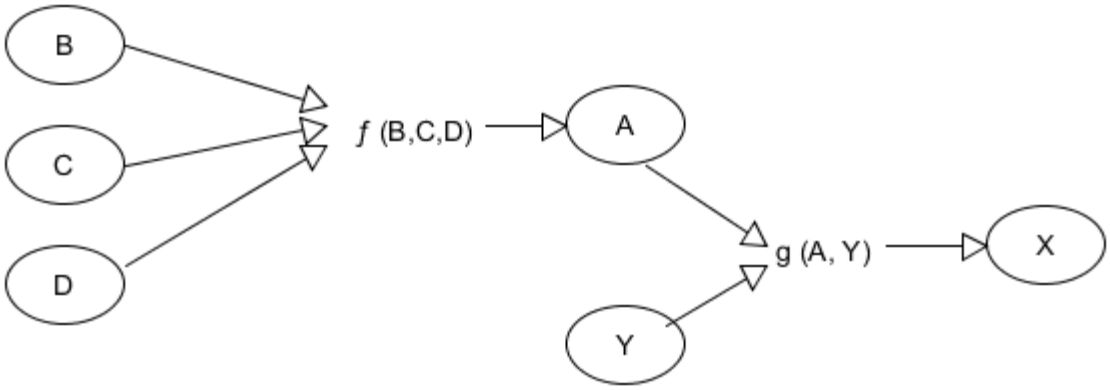


Figure 5: Accurate node relationships

We can here see that the nodes point to a function which again point to the result node. This could be equivalent to the following system of equations:

$$A = f(B, C, D)$$

$$X = g(A, Y)$$

Whenever we change B, C or D, both functions will calculate new values for A and X respectively. If we change Y, only X is computed. One example with function that does nothing, an identity function, and one example without source nodes, a constant function:

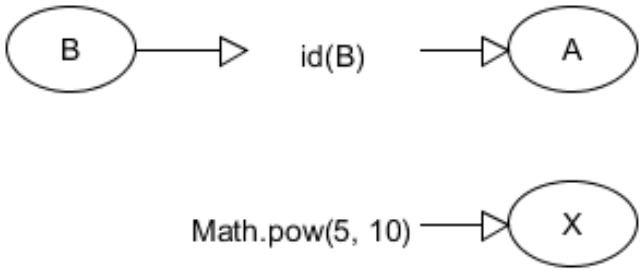


Figure 6: Special cases

This could be equivalent to the following system of equations:

$$A = f(B)$$
$$X = y(5, 10)$$

These are the basic principles and core rules of Hotdrink and its constraint system. There are many other features but it is mostly optimization and techniques that guarantee correctness. These will be discussed in subchapter 2.3 “Internals and significant features of Hotdrink”.

2.2 A simple example

The introduction above to Hotdrink is only about the concepts. We will now take a look at the Hotdrink API and how it is used, for more details see [3].

Starting with dissecting a block of code that implements the easiest example from above, A = B + C:

```
var rules = new hd.ComponentBuilder()
    .variables("A, B, C")
    .constraint("A, B, C")
    .method("B, C -> A",
        (B, C) => B + C)
    .spec();

var component = new hd.Component(rules);
var graph = new hd.PropertyModel();
graph.addComponent(component);
```

First we instantiate a `ComponentBuilder`. This is one of the core objects in Hotdrink. It lets us create a group of business rules. These groups can be connected or disconnected but we will ignore that because it is mostly for optimization.

Next we call a method called `variables()`. This does what you expect it to do, it declares the variables. It takes a string with a comma-separated list of variables as input:

```
var rules = new hd.ComponentBuilder()
    .variables("A, B, C")
```

Most methods return the `ComponentBuilder` object so we can continue calling methods on it and write pretty code without tons of assignments.

Last we call the methods `constraint()` and `method()`. They are connected and have to be called in order with `constraint()` first. The input to `constraint()` is all the variables that will be used in the call to `method()`. This list of variables is then repeated as first parameter to `method()` but with the result variable put at the end after a "->" operator:

```
var rules = new hd.ComponentBuilder()
    .variables("A, B, C")
    .constraint("A, B, C")
    .method("B, C -> A",
```

The framework then knows where you want to put the result. The last input is an anonymous function as the second argument to the `method()` function:

```
.method("B, C -> A",
    (B, C) => B + C)
```

This is the actual function that calculates the values and the same function we described in last subchapter as the dependencies:

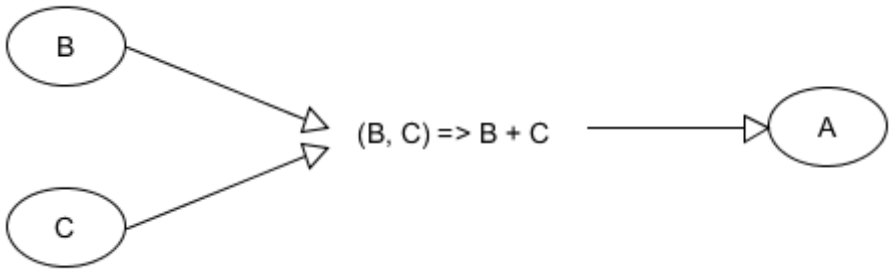


Figure 7: API calls as nodes

The statement `(B,C) => B + C` takes the input variables(B and C) as parameters left of arrow sign "=>" and the calculation on the right. All variables used in the calculation must exist in the parameter list.

In the example above we have a function of the type "arrow function". It could be any JavaScript function, including a function object and a function that returns a function. The latter requires us to put parentheses at the end so we automatically call the method returning the function, see chapter 3.6 "Using native JavaScript to calculating elements before aggregation" for examples.

Finally we call the `spec()` method which initialize the internals after we have entered the rules. To make it all start processing we need to add the rules to a `Component` and then add the component to the `PropertyModel`. The object `PropertyModel` contains the nodes and will trigger the calculations on change.

We also need to bind the nodes in the Hotdrink system to elements on the webpage. This is only necessary if we want to have a demo that we can see and manipulate. This can be done by using simple classes in Hotdrink:

```
function twoWayDomBind(node, htmlObject) {
    var obs = new hd.BasicObservable();
    obs.addObserver(node);
    htmlObject.addEventListener('input', () => {
        obs.sendNext(parseFloat(htmlObject.value));
    });

    node.addObserver({
        onNext: (v) => {
            if (document.activeElement !== htmlObject) {
                htmlObject.value = parseFloat(v).toFixed(2);
            }
        },
        onError: () => {},
        onCompleted: () => {}
    });
}
```

The method `twoWayDomBind()` takes a Hotdrink node and HTML text node as input. In the first block of code we call the method `sendNext()` when we get an event from the input element. The `PropertyModel` will then know that something has changed. In the next block we do the opposite. We replace the method `onNext()` so the HTML element is updated when we receive a new value. This value is then automatically displayed because we are directly manipulating the webpage content via its Document Object Model(DOM).

2.3 Internals and significant features of Hotdrink

The power of Hotdrink comes from its dataflow constraint system [3]. It guarantees that upon completion, we have the correct result because all calculations have been executed in correct order. One of its two most important features is skipping calculations that are no longer needed.

2.3.1 Skipping requests for calculation that are outdated

Assume we have three events, A, B and C. They all set the same variable but use different values, 1, 2 and 3 respectively. We now fire all three events at the same time but in sequence, A, B and C. Normally all three events would be calculated. This is normally not the case with Hotdrink.

First Hotdrink will start executing the first event, A. For the sake of simplicity, we assume that A will demand some computation. While A is processing, Hotdrink will start handling and scheduling the events B and C. It can then identify that all three events change the same variable. A is already processing but B is not started so it can completely skip it because event C will write over the result before it can be read.

This is a nice feature but it strengthens a coupling that is already quite strong in most reactive systems. It requires full control of all calculations and data. Tighter coupling between calculations and data is an important part of this feature's cost. However, this only matters if we want to integrate the system, it is free optimization otherwise.

Although unnecessary coupling is troublesome, this feature is still great because we do not have to handle complex problems like multithreading and race conditions. They can be incredible hard to debug even in simple cases. This is because debugging interrupts normal application flow and may prevent race condition bugs to be recreated.

Leaving the concurrent processing to the framework solves a lot of problems. However, this also means that the framework cannot be extended, wrapped or integrated efficiently with other systems. The requirement for absolute control of state and flow means that from an external system, even simple reading of variables can be very slow. Every get-operation must be a callback because we do not know if the value we ask for is ready. We will discuss this lack of extensibility more in the Evaluation chapter. This is a very important evaluation criteria for my company.

2.3.2 Using and caching optimal execution patterns

Hotdrink can optimize how events are executed. It creates multiple DAGs for each of the source nodes that can receive input. When it receives an event, it will choose the appropriate pathing and execute it. The DAGs are calculated in advanced so executing during runtime can be faster. This is a dynamic optimization of the different calculations the node network will do, depending on what input changes. This is great for complex mathematical calculations where you might need to do experimenting.

The following figure illustrates all possible paths in a simple example. Thick arrows are storing values, dotted arrows are reading:

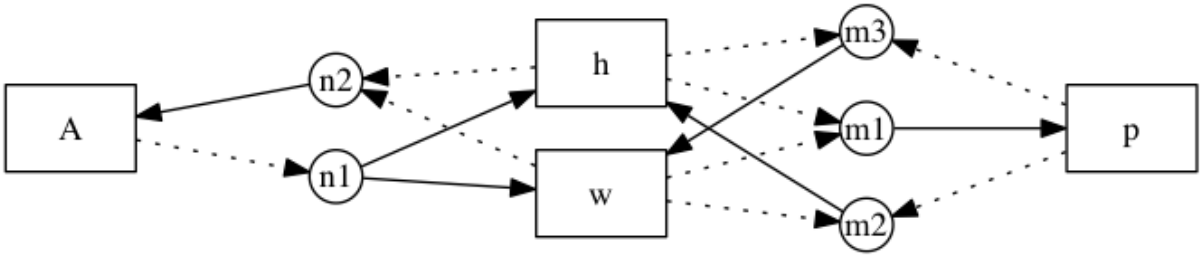
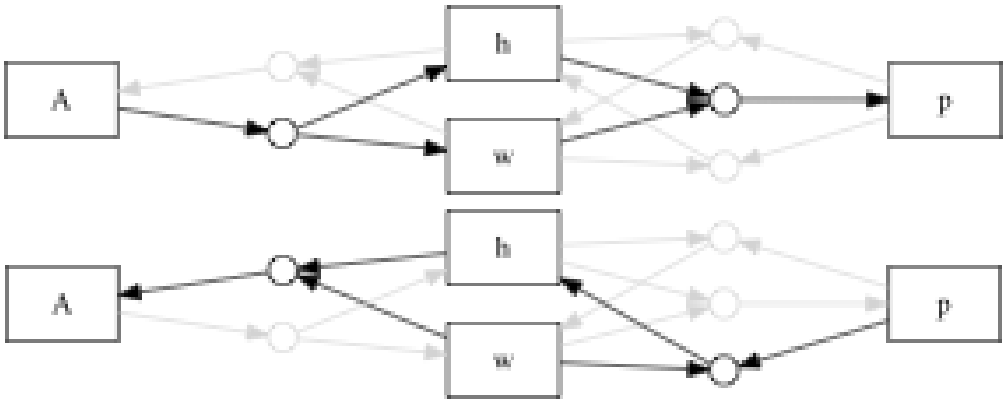


Figure 8: Dependencies when reading and writing

The figure below shows which variables will call which functions and what variables will be changed by that function. Any variable can be changed so for four nodes we will at least have four patterns for calculations, as follows:



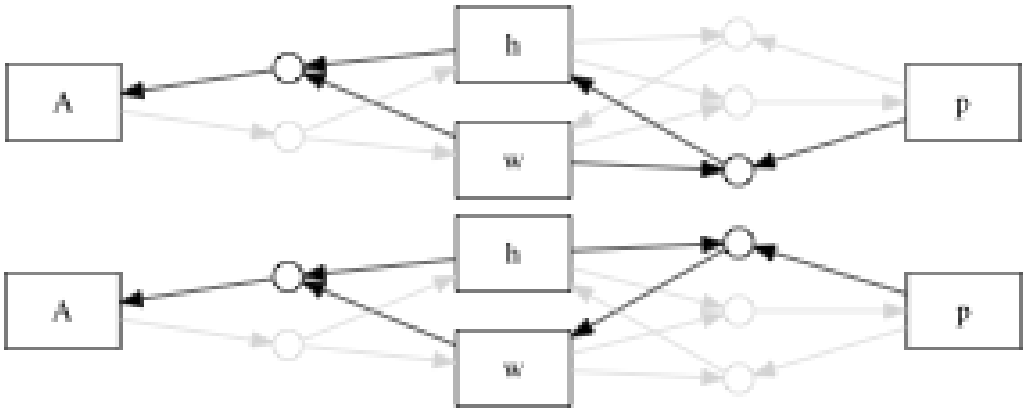


Figure 9: All four possible permutations of Figure 9

Depending on the implementation we can have more patterns because sometimes we do not have to do all calculations in the pattern if one or more results remain unchanged. Unfortunately, this nice feature is quite useless for our financial calculations. In Figure 1: Simplified dependencies, we see that all dependency arrows point in the same direction.

This will be the same for the dimension of time, its dependencies will never go back but only forward. This means that the execution pattern for the calculations never or rarely change for the node network in a financial application. Salary always affects the net worth, net worth never affects salary.

This execution pattern optimization creates quite an overhead and this is especially true for the kind of manipulation we would do in a financial use case. Removing, adding and changing big blocks of data and nodes will create a huge load of work for Hotdrink. This can be optimized by turning the calculations off while we modify it but then we are adding extra state to the calculations which Hotdrink is intended to handle alone. This will be discussed more in the chapter Evaluation 5.3, “Using reactive systems and Hotdrink for finance”.

2.4 Using Hotdrink to solve a financial problem – a naïve approach

My company’s main interest in my master thesis is to see if reactive programming is an effective paradigm for solving financial problems. The company is over 30 years old and has done numerous implementations of the solution which serve as a great reference. Regular imperative programming has been used in all the solutions and the implementations have been done with and without transactions.

By transactions we mean keeping records of the exact transactions. For instance, creating salary transactions for every month for the next 20 years instead of just knowing in advance that you will have a salary at any given month. Transactions shape the application very differently. Because transactions are low level units that are only useful for accountants, they usually become a burden for the expressiveness and business logic in code.

We will now try to solve a basic use case with Hotdrink. It is one of the most regular cases, “How can we analyze a family’s solvency?”. One way to do this is calculating how much is

left of the income after all expenses. The result will affect the net worth, usually indicating a trend for the future net worth, unless there are special circumstances.

The following 2 screen dumps are from my company's software and give us an idea of how it all looks:

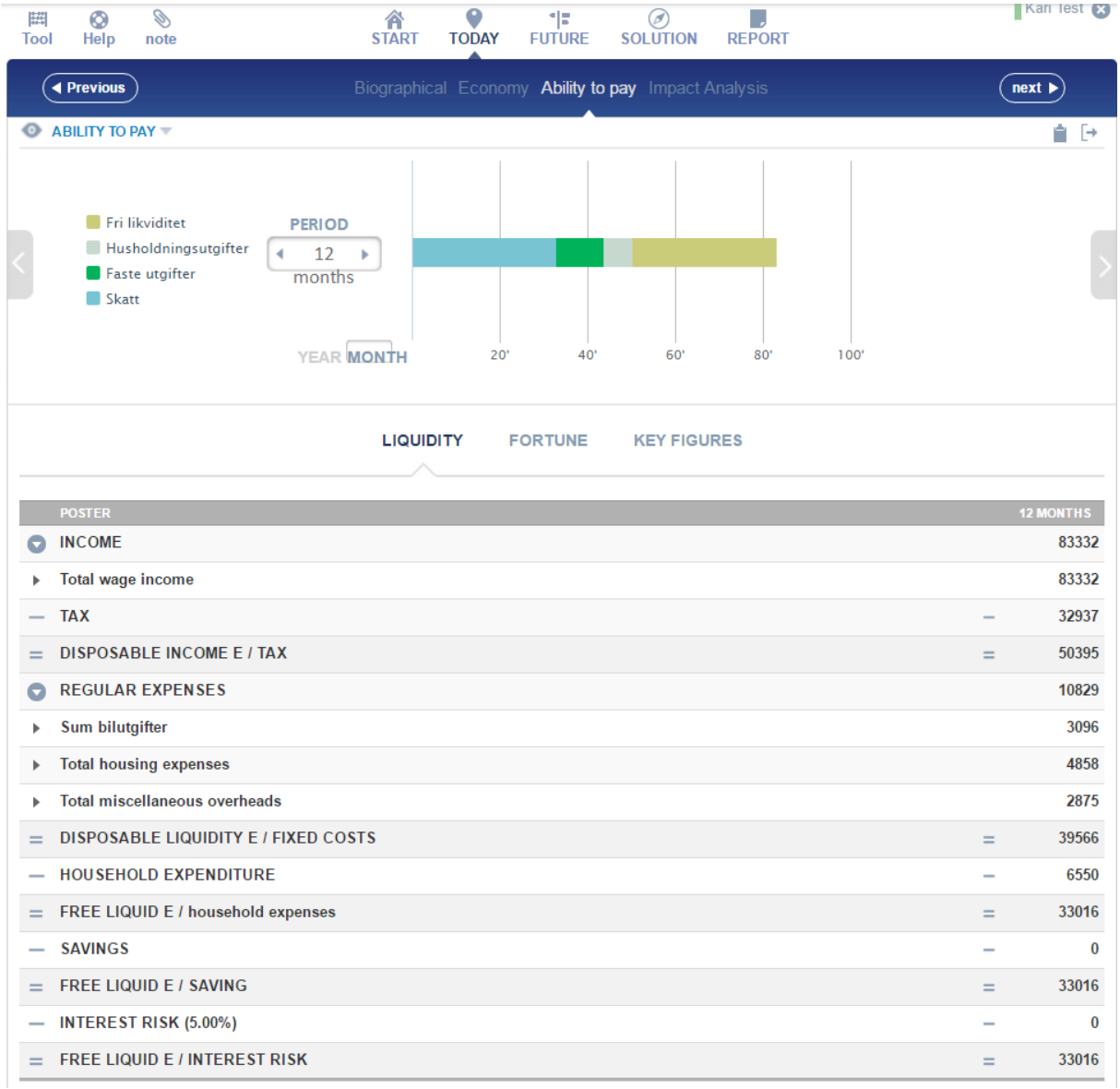


Figure 10: Demo screendump from production version A

This screen is used in production and contain real numbers but with an automatic English translation. In the last line of the table we can see our final answer, how much money is left after all expenses, savings and potential costs from increase in interest rate.

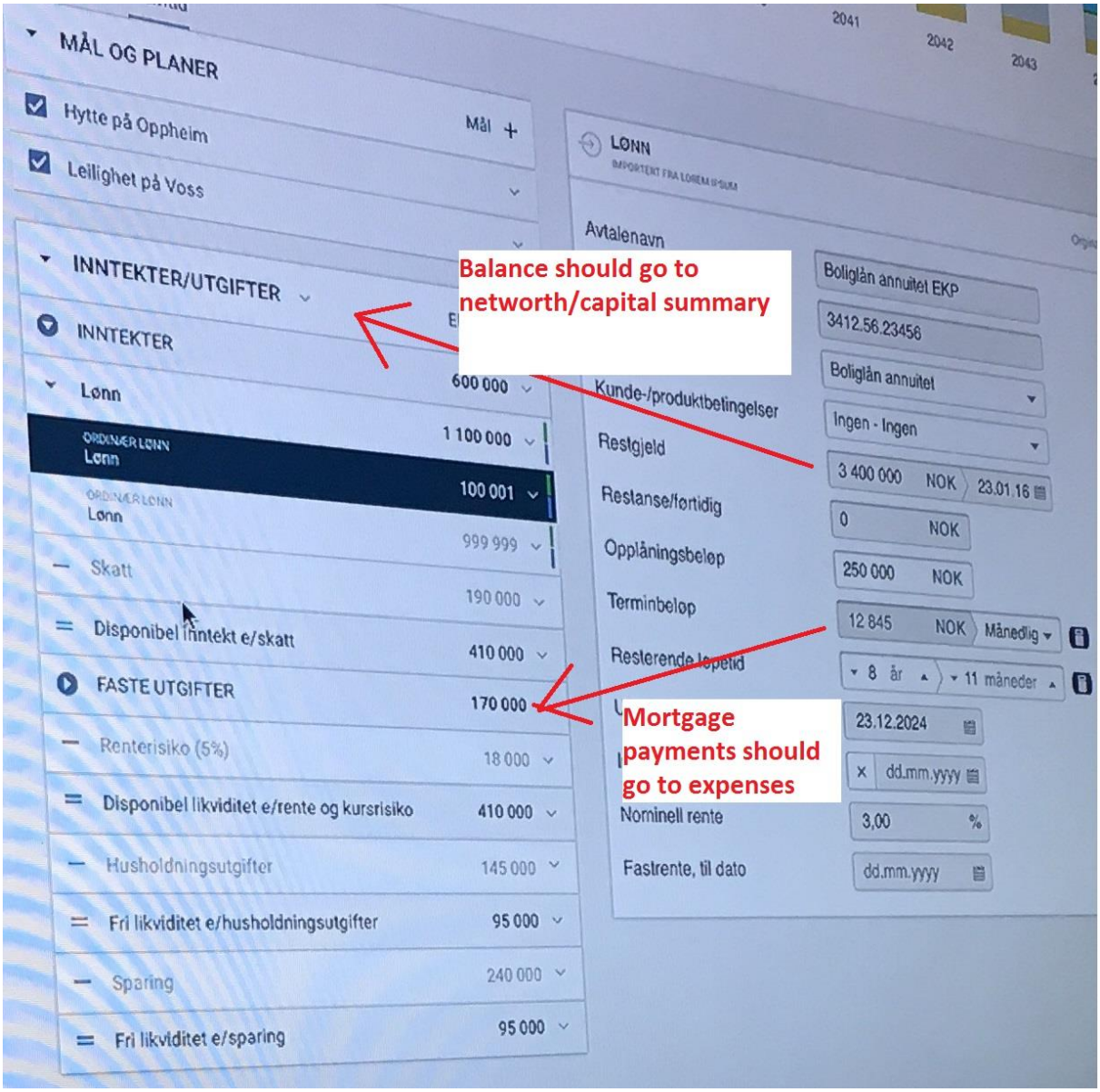


Figure 11: Demo screendump from production version B

This screen is a prototype with fake data and Norwegian text. On the left we can see the overview with the details on the right. This is a typical view in our application, a practical overview on the left and a separate region for details.

The interface should be intuitive after some inspection, even though there is quite a bit of data. Just like a spreadsheet there are accounts like income, expenses, savings, net worth, etc. There are many techniques and approaches to analyze an economy but simplifying it to a single year is very effective. We will do so for this example but will discuss a longer timespan later because it is essential, see section 4.3. If we use a salary and a garage sale as example, the calculations would look like this:

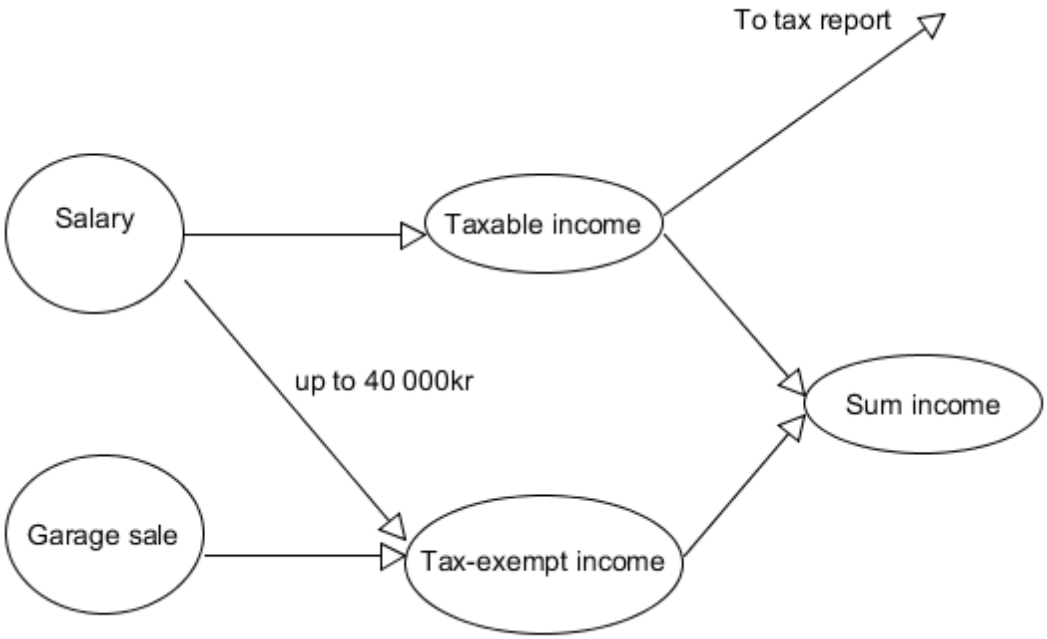


Figure 12: Simple arithmetic except only salary is taxable over 40 000kr

The figure uses only basic arithmetic operations. It is a simple DAG and there are just a few rules. We need to process both the “salary” and the “garage sale” nodes to avoid calculating the “Sum income”-node twice. This is a purely mathematical problem called dependency graph and can thus be solved like one. I will not solve this by myself but use a framework.

Another efficient but also expressive way to solve this mathematical problem in computer science is using a reactive system. Hotdrink is an implementation of such a reactive system which guarantees correct result by executing in the correct order.

We can set up a large number of rules in any order using Hotdrinks API. The system will then solve it for us or refuse it as non-solvable. We do not have to care about dependencies like processing both salary and garage sale before tax-exempt income. This is a huge advantage compared to regular imperative programming where you need to handle all kind of details and side effects manually.

We will now look at an implementation of a more complex Hotdrink example. The API and code can be relatively hard to understand and use. This is because there is a lot of boilerplate code and implicit rules even though it is declarative programming. This is the most important issue I wanted to solve with the code in my master thesis. I wanted the API to be as simple as the mathematics because it did not have to be otherwise. Most of the required parameters in the API are actually implicit from other parameters. We will discuss this more in the next chapter, “Hotdrink as an assembly language”.

In this example, we want to calculate tax from income. The mathematical calculations might look like this:

income = salary + incomeTax + incomeNonTax

tax = salary / 4 + incomeTax / 4

incomeAfterTax = income - tax

In the Hotdrink API it would look like this:

```
var rules = new hd.ComponentBuilder()
    .variables("salary, incomeTax, incomeNonTax, income, tax, incomeAfterTax")
    .constraint("salary, incomeTax, incomeNonTax, income")
        .method("salary, incomeTax, incomeNonTax -> income",
            (salary, incomeTax, incomeNonTax) => salary + incomeTax + incomeNonTax)
    .constraint("salary, incomeTax, incomeNonTax, income,tax")
        .method("salary, incomeTax, incomeNonTax, income -> tax",
            (salary, incomeTax, incomeNonTax, income) => salary / 4 + incomeTax / 4)
    .constraint("salary, incomeTax,incomeNonTax,income,tax,incomeAfterTax")
        .method("salary,incomeTax,incomeNonTax,income,tax -> incomeAfterTax",
            (salary, incomeTax, incomeNonTax, income, tax) => income - tax)
```

This code block executes the exact same logic as the 3 lines of math above. The method `variables()` declares all the variables that exists in the group of rules. `constraint()` declares a rule for some variables. Then we call `method()` on the constraint object with the actual rules. The observant reader might have noticed that we repeat the same variables multiple times. For a given constraint and its method:

```
.constraint("salary, incomeTax,incomeNonTax,income,tax,incomeAfterTax")
    .method("salary,incomeTax,incomeNonTax,income,tax -> incomeAfterTax",
        (salary, incomeTax, incomeNonTax, income, tax) => income - tax)
```

Only 2 parts are logically required:

```
"salary, incomeTax, incomeNonTax, income -> tax",
(salary, incomeTax, incomeNonTax, income) => salary / 4 + incomeTax / 4)
```

We could abstract the whole rule above to the following line of code:

tax = salary / 4 + incomeTax / 4

Now that is something else than 3 lines of long strings with a mix of Javascript operations and parsed strings. This line uses the same syntax as mathematics, which makes it easier to read. To a senior programmer like me, duplication of code is very frustrating and it triggers a

lot of code smell alarms. We always aim to write as little code as possible while still completing the functionality and being expressive.

The example above shows that the API requires a lot of code and is less expressive. This is not to say that it is a bad API, it is just built for mathematical functions and sets of rules that are written manually for correctness and control. In the financial domain we have simpler but a lot more rules compared to the mathematical domain. We have few, if any, cyclic or reversible dependencies. We rarely use systems of linear equations. Generally we follow linear paths of execution through predefined hierarchical structures. An example comparison of typical mathematical and financial logic:

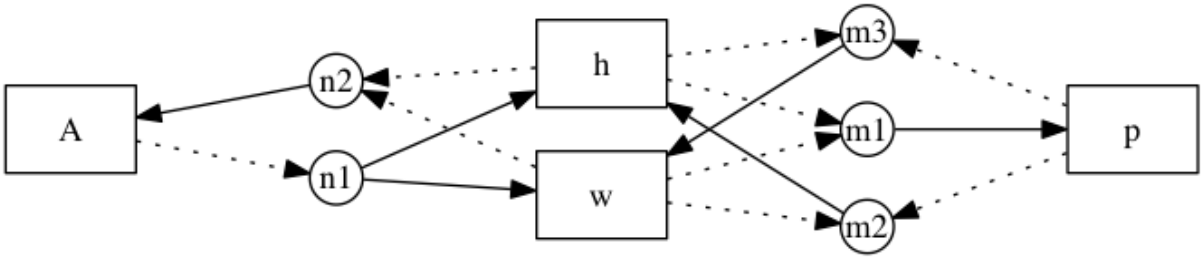


Figure 13: Tree for dependencies in a mathematical domain

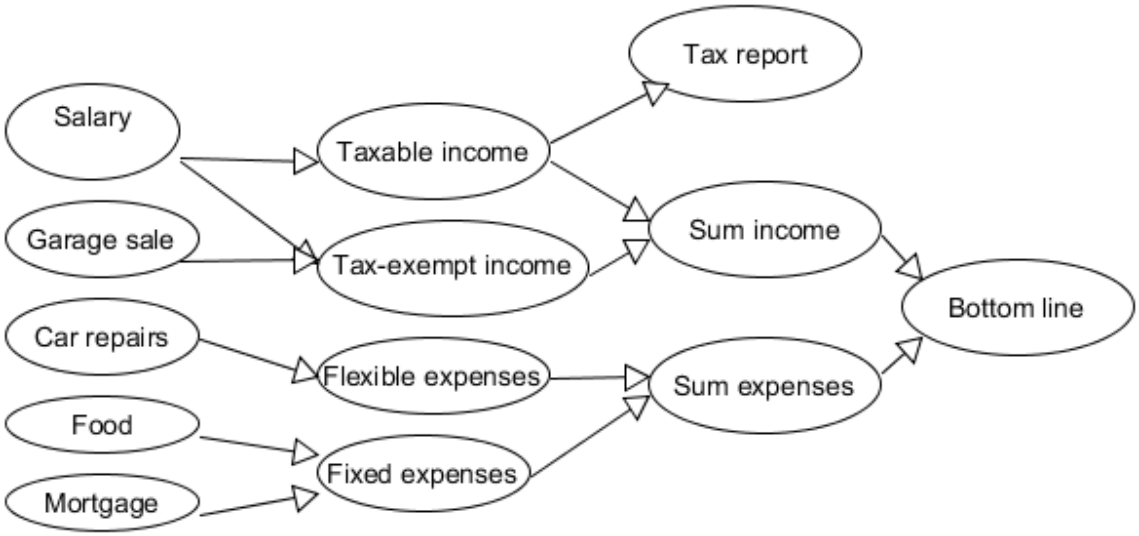


Figure 14: Tree for dependencies in a financial domain

The financial figure contains a lot more data and nodes but is easier to understand because all arrows point in the same direction. The mathematical figure is less intuitive because we do not know where to start and when to end. It also shows why optimizations in pathing are less useful: there are few, if any, alternative paths.

We will now look at a more complex example. We want to underline how verbose and massive a complete application is in the Hotdrink API. The point is not to understand the actual rules but to see how unmanageable the raw API becomes when we have multiple

objects with multiple attributes in our rules. Because there is a lot of data and rules, the example is reduced to 10% of its actual size to fit into the report.

Ignore the macro `FETYPE` for now, it will simply add together all elements of a given type, i.e `INCOME.SALARY` for salaries.
The following semantics:

salary = FETYPE.INCOME.SALARY
*taxes = salary * 0.25*
income = FETYPE.INCOME
bottomLineAfterTax = salary – taxes
livingExpenses = FETYPE.EXPENSES.HOUSEHOLD
allExpenses = FETYPE.EXPENSES
fixedExpenses = allExpenses – livingExpenses

Is equivalent to following declarative code in Hotdrink (some names have been shortened for readability):

```
.variables("OTHERASSETSSHAREDHOUSINGNOeff02bed,OTHERASSETSVEHICLEf9358c11," +
        "INCOMEOTHERca0de7a7,EXPENSESOTHERc4e8435,LOANINSTALLMENTc7cd6c," +
        "OTHERASSETSOWNEDHOUSINGddcc13,EXPENSESHOUSEHOLDbe61378,LOANCREDITc9795," +
        "INCOMESALARYa86b5c,INCOMESALARYef5dd32,EXPENSESMEDICINEHEALTHCAREe213565b," +
        "EXPENSESFUNDSAVINGSAGREEMENTa5d2,EXPENSESTELEPHONEMEDIALEISUREbb794e5," +
        "salary,taxes,income,bottomLineAfterTax,livingExpenses,allExpenses,fixedExpenses")

    .constraint("INCOMESALARYa86b5c,INCOMESALARYef5dd32,salary")

    .method("INCOMESALARYa86b5c,INCOMESALARYef5dd32 -> salary",

        (INCOMESALARYa86b5c, INCOMESALARYef5dd32) => INCOMESALARYa86b5c + INCOMESALARYef5dd32)

    .constraint("salary,taxes")

    .method("salary -> taxes",

        (salary) => salary * 0.25)

    .constraint("INCOMEOTHERca0de7a7,INCOMESALARYa86b5c,INCOMESALARYef5dd32,income")

    .method("INCOMEOTHERca0de7a7,INCOMESALARYa86b5c,INCOMESALARYef5dd32 -> income",

        (INCOMEOTHERca0de7a7, INCOMESALARYa86b5c, INCOMESALARYef5dd32) => INCOMEOTHERca0de7a7 + INCOMESALARYa86b5c + INCOMESALARYef5dd32)

    .constraint("salary,taxes,bottomLineAfterTax")

    .method("salary,taxes -> bottomLineAfterTax",

        (salary, taxes) => salary - taxes)

    .constraint("EXPENSESHOUSEHOLDbe61378,livingExpenses")

    .method("EXPENSESHOUSEHOLDbe61378 -> livingExpenses",

        (EXPENSESHOUSEHOLDbe61378) => EXPENSESHOUSEHOLDbe61378)

    .constraint("EXPENSESOTHERc4e8435,EXPENSESHOUSEHOLDbe61378,EXPENSESMEDICINEHEALTHCAREe213565b" +
        ",EXPENSESFUNDSAVINGSAGREEMENTa5d2,EXPENSESTELEPHONEMEDIALEISUREbb794e5,allExpenses")

    .method("EXPENSESOTHERc4e8435,EXPENSESHOUSEHOLDbe61378,EXPENSESMEDICINEHEALTHCAREe213565b," +
        "EXPENSESFUNDSAVINGSAGREEMENTa5d2,EXPENSESTELEPHONEMEDIALEISUREbb794e5 -> allExpenses",

        (EXPENSESOTHER, EXPENSESHOUS, EXPENSESMEDIC, EXPENSESFUNDSAV, EXPENSESTELE) => EXPENSESOTH + EXPENSESHOUSE + EXPENSESMEDICIN + ..

        .constraint("livingExpenses,allExpenses,fixedExpenses")

        .method("livingExpenses,allExpenses -> fixedExpenses",

            (livingExpenses, allExpenses) => allExpenses - livingExpenses)
```

We can see from the code that we have a lot of long variable names. This is because the data set is from real data. Every attribute of every entity is assigned a generated variable which must to be repeated in all 3 method calls, `variable()`, `constraint()` and `method()`. The example above contains 13 generated variables and 7 regular variables.

A real case would contain at least 10 elements of 3 variables each. More complex cases can contain 20-30 elements of 50 variables each. A range of 30 to 1500 nodes excluding calculations suggests a higher level of abstraction. A calculation of 1500 nodes

would quickly become tens of thousands of lines in the Hotdrink API, even just for simple analyses.

Writing all this code takes a lot of time and is likely to reduce productivity not only during development, but also in debugging and verification. Such horrendous amounts of manually corrected code without tools no longer suggests a higher level of abstraction, it becomes a necessity.

3 Hotdrink as an assembly language

3.1 Higher level of abstraction

The first requirement for the code in my master thesis was to use real data. In our problem domain, real data means a list of elements where each element can have multiple attributes. Abstractions are essential in the financial-economic domain due to the large variety of important and unimportant data. We can say that attributes are abstracted to a single object, objects are abstracted to a list and the list can be in a group of lists.

Hotdrink, on the other hand, is built around variables and functions, not objects. An implementation of arrays exists but it is not documented. Because I was learning reactive programming and Hotdrink on the go, I did not want to add features for objects and attributes over the existing Hotdrink functionality.

Instead I settled on a simpler solution, treating all data as a variable and let a compiler deal with the abstraction of objects and data fetching. This means we need to create a node for every variable used from each element.

The data we are working with is fetched from our internal development server. The data can change at any moment, the unique id's might be replaced and all sorts of cases can be loaded into our Hotdrink code. Because we cannot know the data in advance, we must rely on computing to do it for us.

Abstracting from all the boiler plate code in Hotdrink is very important too. Writing the rules in pure Hotdrink API takes a lot of concentration because most expressions are entered as long strings. The internals of Hotdrink are abstract and very hard to debug if possible at all.

This is a general characteristic of all reactive systems that reach a certain size. All communication is delivered through messages and having thousands of messages makes it very difficult to find the bugs.

One way to ease a lot of these issues is making the API much simpler. Less error prone syntax and code means more focus on business rules and functionality. We do not want to know where a variable is put, how it is treated and what value it contains when. We just want it to be correct and easy to change correctly.

To do this I propose implementing a declarative language over the existing declarative language of Hotdrink. The new language would be compiled to Hotdrink. We can then add the necessary boilerplate code and fetch external data/create data bindings during compilation. This should increase productivity by closing the gap between code and functionality [4]. The code will be more expressive and functionality can be read instead of interpreted.

Because the new declarative language will be customized for financial calculations, it can be considered domain-specific. All its syntax is either a small variation of JavaScript or simply pure JavaScript. We therefore consider it to be a dialect of JavaScript instead of a complete language by itself.

Being domain-specific or a dialect is not important to either the research question or commercialization, it is just an observation. Because we can write close to pure JavaScript in this "dialect", we can use it for any domain or consider it an extension of JavaScript instead of a dialect.

These arguments are merely interpretations and opinions because this thesis is about solving practical problems, and not about programming language theory. We still feel domain-specific dialect is the appropriate name for my work but will refer to it as the “DSL” from here on.

3.2 Creating a domain-specific dialect

The DSL is intended to be readable by non-programmers for verification and development. To populate the node network and solve real scenarios, we need to create macros for fetching data outside the application. This way we can get more useful feedback on solving realistic financial calculations with reactive systems.

Because this is just a master thesis, the amount of time and code is quite limited. By utilizing the powerful features of JavaScript, like `eval()`, we can quickly create a parser without having to deal with low level issues like linking, imports, full syntax validation etc. Combine this with JavaScript being a language with first-class functions, and we can write inline functions straight into the Hotdrink API!

To explain this new DSL we will look at a very simple example. The point is to make a line of the code become a complete Hotdrink API call. First we take the simplest possible line of our new language:

```
salary = 100000
```

The assignment operator does what you expect it to do. The difference between this code and the assignment operator in regular programming languages is that it will just assign a value. Here it will create a function of the constant on the right and put it in the result parameter on the left, “salary”. Because this is a reactive system, the result parameter will be updated and correct at any time assuming there is no calculation currently running. This would generate the following Hotdrink API calls:

```
var rules = new hd.ComponentBuilder()
    .variables("salary")
        .constraint("salary")
        .method(" -> salary",
            () => 100000)
.spec();
```

This code can look a little bit funny because there is only a static number and a variable. This is because Hotdrink does not operate on constants, only nodes and functions. Here it will put the constant as return value of arrow function. It will then assign that function as input to the “salary”-node. Note the almost silly parameters to the function `method()`. The second parameter takes an arrow function with no parameters because only “salary” is affected or used.

Complicating our current example by adding two more variables:

```
salary = 100000
taxRate = 0.25
taxes = salary * taxRate
```

We will now reuse the variables “salary” and “taxRate”. They have both been initialized like in the previous example. The following Hotdrink code is generated:

```
var rules = new hd.ComponentBuilder()
.variables("salary,taxRate,taxes")
 .constraint("salary")
   .method(" -> salary",
     () => 100000)
 .constraint("taxRate")
   .method(" -> taxRate",
     () => 0.25)
 .constraint("salary,taxRate,taxes")
   .method("salary,taxRate -> taxes",
     (salary,taxRate) => salary * taxRate)
.spec();
```

First the method `variables()` contains all three variables. Then we have two constraints similar to the previous example. At last we have a third constraint which use the variables in the first two. Nothing fancy here but now we can see how the input parameters “salary” and “taxRate” are included four times in the last constraint. Our DSL now handles this for us which means less bugs and more efficiency during development.

We will now look briefly at two other features before we move on to the compiler and its macros for fetching external data. First, we have the Math-feature. It would be possible to make the engine allow the following:

```
mathTest = Math.pow(saldoAlleLaan, 10) + 10
```

There is nothing fancy here but to allow this we have to implement a concept of objects, functions, constants, etc. The engine must now understand what an assignment consists of. This is a bit trickier to get right for all cases and becomes almost obsolete after the next feature. Because of this, the Math function is handled as a special case and needs to be assigned alone:

```
saldoAlleLaan = 1000000
mathTest = Math.pow(saldoAlleLaan, 10)
mathTestTwo = mathTest + 10
```

This will generate the following Hotdrink code:

```
.variables("saldoAlleLaan,mathTest,mathTestTwo")
 .constraint("saldoAlleLaan")
   .method(" -> saldoAlleLaan",
     () => 1000000)
 .constraint("saldoAlleLaan,mathTest")
   .method("saldoAlleLaan -> mathTest",
     (saldoAlleLaan) => Math.pow(saldoAlleLaan, 10))
 .constraint("mathTest,mathTestTwo")
   .method("mathTest -> mathTestTwo",
     (mathTest) => mathTest + 10)
```

We see that the JavaScript Math-library can easily be integrated into the Hotdrink API. Now we will take the step the whole way and integrate anonymous functions which give us endless possibilities. Because there is some complexity to parsing code over multiple lines, the parser only supports single line functions in this version. This does not limit the functionality, only the readability. Because readability is important in this thesis, those functions in the DSL will still be written over multiple lines with the exception of the first, more simple example. All generated Hotdrink code will be printed as is.

A simple function can be shown as follows:

```
x = 5
squared = function(x) { return x * x; }
```

We are just squaring a number. The generated Hotdrink code:

```
.variables("x, squared")
  .constraint("x")
    .method(" -> x",
      () => 5)
  .constraint("x,squared")
    .method("x -> squared",
      (x) => function(y) { return y * y; }(x))
```

The first constraint is as expected. The method call of the second constraint is interesting. We create an anonymous function inside the method and call it immediately. This means we can do anything we want whenever the variable “x” changes because we can execute native JavaScript.

Calculating a loan in Hotdrink can be tricky as it usually involves the mathematical operation exponentiation which does not have a native operator in JavaScript. Using the Math-library, we could do it in Hotdrink without using anonymous functions. We could also do with anonymous functions as follows:

```
loanAmountOneMill = 1000000
durationTwentyYearsInMonths = 240
annualInterestRatePercentPoints = 5

monthlyPayment = function (durationTwentyYearsInMonths,loanAmountOneMill,annualInterestRatePercentPoints){
  // convert percentage points to decimal and divide by 12 months
  var monthlyInterestRate = annualInterestRatePercentPoints / 1200;
  return loanAmountOneMill * monthlyInterestRate / (1 - Math.pow(1/(1+monthlyInterestRate),durationTwentyYearsInMonths));
}
```

In this example there is no parsing of the `Math` library. The whole line is identified as an anonymous JavaScript function because it starts with the “function”-keyword right after the assignment-operator. It is then parsed and assigned to a function object which is inserted into the Hotdrink API call:

```
.variables("loanAmountOneMill,durationTwentyYearsInMonths,annualInterestRatePercentPoints,x")
.constraint("loanAmountOneMill")
    .method(" -> loanAmountOneMill",
        () => 1000000)
.constraint("durationTwentyYearsInMonths")
    .method(" -> durationTwentyYearsInMonths",
        () => 240)
.constraint("annualInterestRatePercentPoints")
    .method(" -> annualInterestRatePercentPoints",
        () => 5)
.constraint("loanAmountOneMill,durationTwentyYearsInMonths,annualInterestRatePercentPoints,x")
    .method("loanAmountOneMill,durationTwentyYearsInMonths,annualInterestRatePercentPoints -> x",
        (loanAmountOneMill,durationTwentyYearsInMonths,annualInterestRatePercentPoints) =>
        function (loanAmountOneMill,durationTwentyYearsInMonths,annualInterestRatePercentPoints){
            var monthlyInterestRate = annualInterestRatePercentPoints / 1200;
            return loanAmountOneMill * monthlyInterestRate /
                (1 - Math.pow(1/(1+monthlyInterestRate),durationTwentyYearsInMonths));
     }(loanAmountOneMill,durationTwentyYearsInMonths,annualInterestRatePercentPoints))
```

As we can see above, the Hotdrink code looks quite messy with long variable names. For readability, the code is color coded and we kept the descriptive names.

The calculation above is rather simple. This is because the calculation ignores factors like fees, leap years, real accuracy for different length of months, no decimals, etc. To implement a loan calculation with those requirements we must do it iteratively. Creating a formula is theoretically possible but, a lot of corner cases and changes would make it impractical and unmaintainable.

We will now look at the last example of the anonymous functions which uses a loop to calculate the length of a loan. It takes the size of the loan, maximum monthly payment, interest rate and a fee as input. This is difficult to solve with other methods such as mathematical formulas. It provides us with a good example of how reactive systems are flexible and easy to integrate to a certain point. The DSL code with an anonymous function:

```
loan = 1000000
monthlyPayment = 7000
rate = 5
fee = 75
duration = function(loan, monthlyPayment, rate, fee) {
  var monthlyInterestRate = rate / 1200;
  var month = 0;
  while (loan > 0) {
    var interest = loan * monthlyInterestRate;
    var principal = monthlyPayment - interest - fee;

    if (principal > loan) {
      principal = loan;
      loan = 0;
    } else {
      loan -= principal;
    }
    month++;
  }
  return month;
}
```

The code is quite self-explanatory, it ends when there is no more loan to pay. It then returns the duration of the loan. This is a weakness of the reactive system. It does not operate on objects so you would have to calculate everything a second time to get other important results from a duration calculation, e.g. total cost of the loan. The resulting Hotdrink code is as follows:

```
.variables("loan,monthlyPayment,rate,fee,duration")
  .constraint("loan")
  .method(" -> loan",
    () => 1000000)
  .constraint("monthlyPayment")
  .method(" -> monthlyPayment",
    () => 7000)
  .constraint("rate")
  .method(" -> rate",
    () => 5)
  .constraint("fee")
  .method(" -> fee",
    () => 75)
  .constraint("loan,monthlyPayment,rate,fee,duration")
  .method("loan,monthlyPayment,rate,fee -> duration",
    (loan, monthlyPayment, rate, fee) => function(loan, monthlyPayment, rate, fee) {
      var monthlyInterestRate = rate / 1200;
      var month = 0;
      while (loan > 0) {
        var interest = loan * monthlyInterestRate;
        var principal = monthlyPayment - interest - fee;
        if (principal > loan) {
          principal = loan;
          loan = 0;
        } else {
          loan -= principal;
        }
        month++;
      }
      return month;
    }(loan, monthlyPayment, rate, fee))
```

The anonymous function used in the generated Hotdrink code is the exact same code as in the DSL, except the input arguments are replaced with actual variables. We cannot see the replacement in this example because the actual variables used have the same names as the arguments. We will see this later with macros and dynamic variables in chapter 3.6 “Using native JavaScript to calculating elements before aggregation”.

The code we execute in anonymous functions is identical to the code in the DSL. This makes the code easier to develop, verify and maintain because it is completely isolated from its dependencies.

However, it is quite clear that depending on anonymous functions for most use cases is not the solution. It is more technical, verbose, and the native JavaScript will execute everything whenever one of its input parameters changes. This is unlike Hotdrink’s own calculation which will optimize it. Furthermore, complexity inside complexity is generally discouraged and this feature is only intended for cases that cannot be solved otherwise.

3.3 Parsing statements

We will now describe how parsing of statements is done. The full process is considered a variation of compilation, but the first step in compiling is parsing and thus we use the words parser and parsing for most of this section. We will comment on this at the end.

The simplest parsing it can do are expressions with only arithmetic operations and variables. It takes a line like “`INCOME = salary + 10000`”, and splits into 2 by the equal sign. It then uses the left side of the equal sign as result variable. The statement on the right side is then parsed as a separate expression.

It will look for certain keywords like “Math” or “function” when parsing the right side expression. If no keyword is found, it will parse all parts split by the character space (“ “).

Any part that is not an arithmetic operator or a number is considered a variable. It will be added to the list of variables.

When we are done parsing everything on the right side of the assignment operator we can generate a function. This is the function that will bind the separate variables together with a "reactive dependency". Whenever "`salary`" changes, "`INCOME`" will be automatically updated by this function with "`salary`" as input.

The parameter part of the function is on the left side of the arrow operator("=>"). It will consist of all variables used in the expression but comma separated:

```
(salary) =>
```

Another incomplete example:

```
(salary, income,loan, networth) =>
```

This code is pure JavaScript and follow JavaScript rules. Variable names do not have to be in order because they must match the right side of the expression. The variables names also have to follow JavaScripts standard, meaning no leading numbers, special characters etc.

After we build the left side we will complete the right side. This is usually just replacing the whole statement from the initial right side, in our example "salary + 10000". However, if we have macros in the statement, we have to remove all the macro commands. They are replaced with references to the variables that are generated from the data, depending on the type of macro. More details and examples with macros can be found in the next subchapter, 3.4 "Retrieving external data".

When we have added the statement for the right side it will look like this:

```
(salary) => salary + 10000
```

This string containing a statement is then sent to the `eval()`-method in JavaScript to create a function object from the string. We then generate the Hotdrink API expressions which is basically just a sequence of duplication and simple mapping:

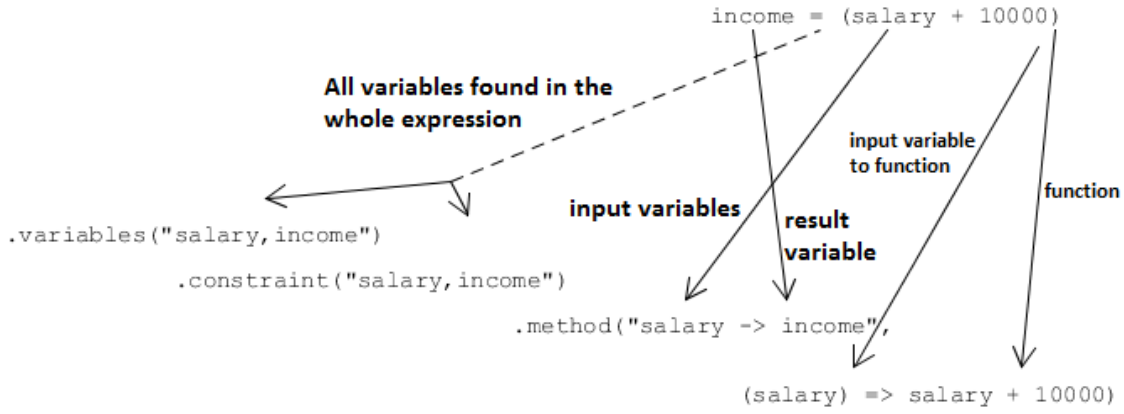


Figure 15: Parsing the DSL and mapping to the API

In the figure we can see how many times "salary" and "income" are written even though the initial and logic expression only does it once. This example is with predefined variables. Generated variables are much longer and messier. This can be seen in code examples from next subchapter.

The parser does not actually generate executable code. The parameters are generated and then inserted into the API calls. It will print the rules that are executed in the console as debug information. This can then be copied to a Hotdrink program and executed for verification or manual modification.

Creating a parser from scratch in limited time means that many corner cases and irregular programming rules does not work in this language.
For example, the following statement "INCOME = SALARY = 10000", is perfectly valid in most languages but rarely used. This would generate a parse error in my parser because it expects only 1 equal sign unless it is inside an anonymous function.

The expression "`A = Math.pow(10,10) + 10`" will not work either because it is a special case together with "+ 10". It must be written on 2 separate lines to work:

```
A = Math.pow(10,10)
B = A + 10
```

Nor does it support overwriting variables:

```
A = Math.pow(10,10)
A = A + 10
```

This is quite normal in regular imperative programming but these variables are actually nodes with automatic calculations. The expression "A = A + 10" would create an infinite loop but Hotdrink will identify it and throw an error. Whenever "exponent" is changed, we should perform the calculation and store the result in "exponent", triggering the cycle.

Some of these issues are just scoped out because of time. They are not important to the solution and merely extra features that we expect from complete languages. The last issue with infinite loops already have a technical solution.

A note on parsing, compiling and their variations. The code works like a compiler although it is a lot simpler than most compilers. To be more accurate, it compiles the new DSL to Hotdrink API expressions which are executed immediately when done parsing.

Because Hotdrink and its API is executed as a high-level language(JavaScript) the compiler is technically a source-to-source compiler. Transpiler and transcompiler are others names for this kind of transformation. None of this really matters for the end result because it is still executed as JavaScript. Being executed as JavaScript means it will be interpreted by the browser during runtime. The interpreted code will contain no low-level code, neither platform specific nor platform independent like bytecode. For simplicity throughout this text, we will use the word compiler.

3.4 Retrieving external data

Earlier we looked at the declarative language and its parser. Now we will look at the macros that make the system work. Until now we had to type in everything. My company already has a lot of real data and using it is a requirement for the thesis. The first macro implemented was the “FETYPE”. These macros are solely based on the data structure in our bank production system and most convenient features have been excluded because of the tight thesis scope. The horribly long names for generated variables are an example of this.

We will for now assume that a financial element can only have one value. By financial element we mean a part of the economy, whether it is a child, wife, car, golf expenses, house or savings agreement. These are factors that can have costs, income and interest. We usually calculate a single household and this means we can have multiple salaries for multiple persons. Differentiating on persons is only required for more advanced analysis so we have removed the owner-dimension from the scope. To show the sum of salaries we need to make the macro perform *addition of all salary elements*.

To identify the salary elements, we use a property called “modelUUID”. This property tells us which kind of object we have. The potential keys follow a logic structure like “INCOME.SALARY”, “EXPENSES.RENTAL.HOUSING” or “INSURANCE.CRITICAL.ILLNESS”. Because the all the models follow the notation with a dot after its category, we can identify all incomes or expenses easily.

Note, they will not identify models that can be considered more than one category. An example of this is housing. A primary home is categorized as an asset but renting is categorized as an expense. From a private customer’s point of view this is often completely irrelevant, it is just what we pay for a roof. The business logic must explicitly handle all these variations depending on the business case.

The following commands will give us total income, sum of salaries and the non-salary income:

```
totalIncome = FETYPE.INCOME
sumSalaries = FETYPE.INCOME.SALARY
incomeNonSalary = totalIncome – sumSalaries
```

The generated Hotdrink code:

```
.variables("INCOMECOMPANYCARc8534279,INCOMESALARYb,totalIncome,sumSalaries,incomeNonSalary")
    .constraint("INCOMECOMPANYCARc8534279,INCOMESALARYb,totalIncome")
        .method("INCOMECOMPANYCARc8534279,INCOMESALARYb -> totalIncome",
            (INCOMECOMPANYCARc8534279, INCOMESALARYb) => INCOMECOMPANYCARc8534279 + INCOMESALARYb)
    .constraint("INCOMESALARYb,sumSalaries")
        .method("INCOMESALARYb -> sumSalaries",
            (INCOMESALARYb) => INCOMESALARYb)
    .constraint("totalIncome,sumSalaries,incomeNonSalary")
        .method("totalIncome,sumSalaries -> incomeNonSalary",
            (totalIncome, sumSalaries) => totalIncome - sumSalaries)
```

The fetched dynamic data for this example is a salary and the taxable benefit of a company car(considered income). We see that the generated code contains two new variables that are not in the declarative code, "`INCOMECOMPANYCARc8534279`" and "`INCOMESALARYb`". These are the generated nodes from the data set. The variable names themselves are simply the "modelUUID" concatenated with the "UUID" of that element.

The property "modelUUID" is used for readability, so we can see what elements we have. The property "UUID" is the unique id for any given element and used to ensure a unique id for the variable node in Hotdrink.

Because the variable names will later be used as JavaScript variables, they must cohere with the JavaScript variable requirements. The dots in the "modelUUID" and any dash in the "UUID" are removed. We can also see a bug in the naming where all other characters but "b" have been removed in "`INCOMESALARYb`".

We can now use generated data in our DSL and node network but everything is executed like before. The generated nodes are added like predefined variables. The only difference is the how the content is set. In the code above we see that there is no constants or values like in the earlier examples. This is because Hotdrink can set up a network and then later populate it with actual data. When we used constants earlier we say that variable X is the result of a function that always returns the constant Y.

Now with macros we will insert the values from the data set. We do this by calling the `set()`-method on the nodes that only are sources, also known as leaf or source nodes. This is done after the network is built, in the same way you would manipulate it during use. The data could be inserted into the Hotdrink API before execution but then the coupling between logic and data would be tighter. We generally always prefer low coupling unless it is necessary or a natural relation in the domain.

3.5 Macro for using detailed data

Until now we have only looked at a macro that fetches a single type of elements and add them together. Most calculations require more than a single value or only aggregated values. If we would like to calculate the estimated total costs for the cars, we would need to use both the car model and registration year for all the cars separately. We will solve this problem by extending our existing macro "FETYPE".

We extend the macro "FETYPE" with a syntax "prop" at the end, short for "properties". We will explain it with a previously mentioned case. In the problem with assets we want to find the calculated cost of cars, not their value. If we write:

cars = FETYPE.OTHERASSETS.VEHICLE

The variable "cars" will contain the sum of the market value of all the cars. This could be relevant but we want the costs of owning the car(s). The total yearly costs for a car is stored in the property "sumExpensesFields". With the new macro, we can then write:

totalCostsOfCars = FETYPE.OTHERASSETS.VEHICLE.prop.sumExpensesFields

This will result in the following Hotdrink code(here with extra formatting):

```
.variables("OTHERASSETSSHAREDHOUSINGNOeff02bed," +
    "OTHERASSETSVEHICLEf9358c11, " +
    "INCOMEOTHERca0de7a7," +
    "EXPENSESOTHERc4e8435, " +
    "LOANINSTALLMENTc7cd6c, " +
    "OTHERASSETSOWNEDHOUSINGddcc13," +
    "EXPENSESHOUSEHOLDbe61378, " +
    "LOANCREDITc9795," +
    "INCOMESALARYa86b5c, " +
    "INCOMESALARYef5dd32, " +
    "EXPENSESMEDICINEHEALTHCAREe213565b," +
    "EXPENSESFUNDSAVINGSAGREEMENTa5d2, " +
    "EXPENSESTELEPHONEMEDIALEISUREbb794e5," +
    "totalCostsOfCars, " +
    "OTHERASSETSVEHICLEf9358c11PROPsumExpensesFields")
  .constraint("OTHERASSETSVEHICLEf9358c11," +
    "totalCostsOfCars, " +
    "OTHERASSETSVEHICLEf9358c11PROPsumExpensesFields")
  .method("OTHERASSETSVEHICLEf9358c11," +
    "OTHERASSETSVEHICLEf9358c11PROPsumExpensesFields - > totalCostsOfCars ",
    (OTHERASSETSVEHICLEf9358c11, OTHERASSETSVEHICLEf9358c11PROPsumExpensesFields) =>
          OTHERASSETSVEHICLEf9358c11PROPsumExpensesFields)
```

In the last lines we see a new type of variable that is very long. This variable contains PROP in the middle of it, after the unique identifier, "OTHERASSETSVEHICLEf9358c11", and before the property we want to use, "sumExpensesFields". There is no reason for this variable to be this long other than readability but that is also an important part of development. This macro solves multiple cases and requirements. For example, on the tax report we need to put in how many kilometers we drive. We could simply write:

drivingToWork = FETYPE.OTHERASSETS.VEHICLE.prop.annualDrivingDistance

drivingAtWork = FETYPE.INCOME.COMPANYCAR.prop.YrkeskjoeringAntKm

sumYearlyDrivingDistance = drivingToWork + drivingAtWork

It would be nice to have this in one line but this is not yet supported because of the scope. It does not add value as a proof of concept and in this exact case it clarifies the Norwegian name for the property "YrkeskjoernigAntKm" used in production.

Another example is, how much total area are all my housing properties?

ownedHousingArea = FETYPE.OTHERASSETS.OWNEDHOUSING.prop.usageAcreage

cooperativeHousingArea = FETYPE.OTHERASSETS.COOP.APARTMENT.prop.usageAcreage

totalArea = ownedHousingArea + cooperativeHousingArea

3.6 Using native JavaScript to calculating elements before aggregation

We are still unable to solve our loan costs problem with the features presented so far. To calculate the costs correctly, we need perform the loan calculation in isolation for each element. Only then can we achieve the correct result by adding them together **after** the complete calculation.

To do this we combine the FETYPE-PROP-macro with anonymous functions. We will reuse the example from earlier in this chapter where we calculated the length of a loan. The example without macros was:

```
loan = 1000000
monthlyPayment = 7000
rate = 5
fee = 75
duration = function(loan, monthlyPayment, rate, fee) {
  var monthlyInterestRate = rate / 1200;
  var month = 0;
  while (loan > 0) {
    var interest = loan * monthlyInterestRate;
    var principal = monthlyPayment - interest - fee;

    if (principal > loan) {
      principal = loan;
      loan = 0;
    } else {
      loan -= principal;
    }
    month++;
  }
  return month;
}
```

If we replace the variables with macros we get the following:

```
duration = function(FETYPE.LOAN.INSTALLMENT.prop.value, FETYPE.LOAN.INSTALLMENT.prop.amount,
                    FETYPE.LOAN.INSTALLMENT.prop.nominalInterestRate, FETYPE.LOAN.INSTALLMENT.prop.instalmentFee) {
  var monthlyInterestRate = nominalInterestRate / 1200;
  var month = 0;
  while (value > 0) {
    var interest = value * monthlyInterestRate;
    var principal = amount - interest - instalmentFee;
    if (principal > value) {
      principal = value;
      value = 0;
    } else {
      value -= principal;
    }
    month++;
  }
  return month;
}
```

Note that the variables inside the functions have changed a little. This is because the compiler removes the macro-syntax and only the property itself is left. For example, the

macro “FETYPE.LOAN.INSTALLMENT.prop.**nominalInterestRate**” is only referred to as “nominalInterestRate” inside the function.

Before we had the variables “loan” and “monthlyPayment” inside the function. Now they are replaced with the variables “value” and “amount” because those are the actual property names where the data is stored. This is a degradation of readability but could be solved rather easily with more time because it is only manipulation of strings. Then we would have the same name used inside the function regardless of the data source. It is also a step towards reusable custom functions.

The generated Hotdrink code:

```
.variables("duration,LOANINSTALLMENTa380ef01PROPvalue,L..PROPamount,L..PROPnominalInterestRate,L..PROPinstalmentFee")
.constraint("LOANINSTALLMENTa380ef01PROPvalue")
  .method(" -> LOANINSTALLMENTa380ef01PROPvalue",
    (LOANINSTALLMENTa380ef01PROPvalue) => 1000000)
.constraint("LOANINSTALLMENTa380ef01PROPamount")
  .method(" -> LOANINSTALLMENTa380ef01PROPamount",
    (LOANINSTALLMENTa380ef01PROPamount) => 5596)
.constraint("LOANINSTALLMENTa380ef01PROPnominalInterestRate")
  .method(" -> LOANINSTALLMENTa380ef01PROPnominalInterestRate",
    (LOANINSTALLMENTa380ef01PROPnominalInterestRate) => 3)
.constraint("LOANINSTALLMENTa380ef01PROPinstalmentFee")
  .method(" -> LOANINSTALLMENTa380ef01PROPinstalmentFee",
    (LOANINSTALLMENTa380ef01PROPinstalmentFee) => 50)
.constraint("duration,LOANINSTALLMENTa380ef01PROPvalue,L...PROPamount,L...PROPnominalInterestRate,L...PROPinstalmentFee")
  .method("LOANINSTALLMENTa380ef01PROPvalue,L...PROPamount,L...PROPnominalInterestRate,L...PROPinstalmentFee -> duration",
    (LOANINSTALLMENTa380ef01PROPvalue, L..PROPamount, L..PROPnominalInterestRate, L..PROPinstalmentFee) =>
        function(value, amount, nominalInterestRate, instalmentFee) {
          var monthlyInterestRate = nominalInterestRate / 1200;
          var month = 0;
          while (value > 0) {
              var interest = value * monthlyInterestRate;
              var principal = amount - interest - instalmentFee;
              if (principal > value) {
                  principal = value;
                  value = 0;
              } else {
                  value -= principal;
              }
              month++;
          }
          return month;
    }(LOANINSTALLMENTa380ef01PROPvalue, LOANINSTALLMENTa380ef01PROPamount,
      LOANINSTALLMENTa380ef01PROPnominalInterestRate, LOANINSTALLMENTa380ef01PROPinstalmentFee))
```

Note the syntax with multiple dots “L…PROPamount”. It is only a replacement to make the example with long names readable.

All these tools and features allow us to solve a lot of use cases. It improves readability and is usually easy to debug. The debugging is easier because we can isolate the buggy parts of the code and only focus on a single part at a time. The current version of the DSL can be too verbose at times because it does not accept multiple statements on a single line other than arithmetic operations. This is ok for now because we see it more as a possible improvement than a critical feature for the proof of concept.

4 Proposal for financial calculations using reactive programming

The main research problem for my thesis is “Give a proof of concept for financial calculations using reactive programming. It will be evaluated on readability, maintainability, extensibility and efficiency.”

Originally the main research problem included a dimension of time. After some research and discussions with my colleagues this changed. It was too difficult to implement a dynamic accuracy for time, switching between time-scales such as day, month and year, all within a few months of work. To gain any practical value from a master thesis, we had to skip this complicated feature.

A dimension of time only makes sense with the other two dimensions, input data and posting entries to accounts. The last two dimensions are much easier because they are well known and fit nicely into a tree structure. Of course, the flow of time is also known but it affects all the different financial elements in different ways. This makes it hard to standardize on a structure which represents the data over time.

It is simply too complex because any simplification, like only years, months or days, is not useful. It is also overlaps closely with another complex issue, aggregation. More on this can be found in section 4.3.

4.1 Solution for analyzing the current solvency while ignoring the future

The closest possible problem that I could propose a solution for was the following:
Use the declarative language to analyze solvency, net worth and tax for the current year. This is easier and achievable because all stored values are for the current year or for an average month in the current year.

We will first present all the DSL code in the proposed solution before we explain. Hopefully this gives the reader a more accurate impression of the readability, or the lack thereof.

The different categories of calculations begin with a comment line like "`//Income`":

```
//Income
regularSalaries = FETYPE.INCOME.SALARY
miscellaneousSalaries = FETYPE.INCOME.OTHER
totalIncomes = FETYPE.INCOME

//Expenses
householdExpenses = FETYPE.EXPENSES.HOUSEHOLD
carExpenses = FETYPE.OTHERASSETS.VEHICLE.prop.sumExpensesFields
housingExpenses = FETYPE.OTHERASSETS.OWNEDHOUSING.prop.sumExpensesFields
installmentLoanMonthlyExpenses = function(FETYPE.LOAN.INSTALLMENT.prop.value, FETYPE.LOAN.INSTALLMENT.prop.durationMonths,
                                          FETYPE.LOAN.INSTALLMENT.prop.nominalInterestRate,
                                          FETYPE.LOAN.INSTALLMENT.prop.instalmentFee) {
    var amountThreshold = 100;
    var iterationLimit = 100;
    if (!instalmentFee) {
        instalmentFee = 0;
    }
    var calculate = function(value, duration, amount, nominalInterestRate, instalmentFee) {
        var monthlyInterestRate = nominalInterestRate / 1200;
        var month = 1;
        var finished = function(value) {
            return (value < 0 && Math.abs(value) < amountThreshold) || (value > 0 && value < amountThreshold);
        };
        var interest;
        var monthlyMortgage;
        while (month <= duration) {
            interest = value * monthlyInterestRate;
            monthlyMortgage = amount - interest - instalmentFee;
            value -= monthlyMortgage;
            if (finished(value)) {
                return amount;
            }
            if (value < 0) {
                return -1;
            }
            month++;
        }
        return 0;
    };
    var interpolation = function(value, duration, nominalInterestRate, instalmentFee) {
        var amount = 0;
        var lowest = value / duration;
        var highest = value * 0.5 * (duration / 12);
        var iteration = 0;
        var pivotAmount;
        var result;
        while (amount === 0 && iteration < iterationLimit) {
            pivotAmount = (highest + lowest) / 2;
            result = calculate(value, duration, pivotAmount, nominalInterestRate, instalmentFee);
            if (result === 0) {
                lowest = pivotAmount;
            } else if (result === -1) {
                highest = pivotAmount;
            } else {
                return result;
            }
            iteration++;
        }
    };
    return interpolation(value, durationMonths, nominalInterestRate, instalmentFee);
};
yearlyCreditExpenses = installmentLoanMonthlyExpenses * 12 + creditLoanExpenses
miscellaneousExpenses = FETYPE.EXPENSES
savings = FETYPE.EXPENSES.FUND.SAVINGSAGREEMENT
fixedExpenses = miscellaneousExpenses - householdExpenses + housingExpenses + yearlyCreditExpenses + carExpenses - savings

//Assets
assets = FETYPE.OTHERASSETS
debt = FETYPE.LOAN
networth = assets - debt
```

```
//Taxes
debtInterestDeductionRatePercentage = 25
nationalInsuranceContributionsRatePercentage = 7.8
grossIncome = regularSalaries + miscellaneousSalaries
personalAllowance = 81000
basicAllowance = 47150
yearlyLoanInterest = function(FETYPE.LOAN.INSTALLMENT.prop.value, FETYPE.LOAN.INSTALLMENT.prop.amount,
                              FETYPE.LOAN.INSTALLMENT.prop.durationMonths, FETYPE.LOAN.INSTALLMENT.prop.nominalInterestRate,
                              FETYPE.LOAN.INSTALLMENT.prop.instalmentFee) {
    var monthlyInterestRate = nominalInterestRate / 1200;
    var month = 1;
    sum = 0;
    var monthlyMortgage = 0;
    while (month <= durationMonths && month <= 12) {
        interest = value * monthlyInterestRate;
        sum += interest;
        monthlyMortgage = amount - interest - instalmentFee;
        value -= monthlyMortgage;
        month++;
    }
    return sum;
};

debtInterestDeduction = (debtInterestDeductionRatePercentage / 100) * (creditLoanExpenses + yearlyLoanInterest)
generalIncome = grossIncome - personalAllowance - debtInterestDeduction - basicAllowance

nationalInsuranceContribution = generalIncome * (nationalInsuranceContributionsRatePercentage / 100)
wealthTaxRatePercentage = 0.7
wealthTax = function(networth, wealthTaxRatePercentage) {
    if (networth >= 1480000) {
        return networth * (wealthTaxRatePercentage / 100);
    } else {
        return 0;
    }
}

municipalCountyStateTaxRatePercentage = 28
municipalCountyStateTax = generalIncome * (municipalCountyStateTaxRatePercentage / 100)

sumTax = municipalCountyStateTax + wealthTax + nationalInsuranceContribution
incomeAfterTax = totalIncomes - sumTax


//Interest risk
sumCredits = FETYPE.LOAN.CREDIT
sumLoans = FETYPE.LOAN.INSTALLMENT
sumDebt = totalCredits + totalLoans
interestRateRiskPercentage = 5
interestRiskBeforeTax = sumDebt * (interestRateRiskPercentage / 100)
interestRateRiskAfterTax = interestRiskBeforeTax * (1 - debtInterestDeductionRatePercentage)


//Solvency
solvencyAfterFixedExpensesAndInterestRisk = incomeAfterTax - fixedExpenses - renterisiko
solvencyAfterLivingExpenses = solvencyAfterFixedExpensesAndInterestRisk - householdExpenses

solvencyAfterSavings = solvencyAfterLivingExpenses - savings

expectedPropertyValueIncreasePercentage = 5
ownedProperties = FETYPE.OTHERASSETS.OWNEDHOUSING
cooperativeProperties = FETYPE.OTHERASSETS.SHAREDHOUSING.NO
currentValueProperties = ownedProperties + cooperativeProperties
propertiesValueIncrease = currentValueProperties * expectedPropertyValueIncreasePercentage

expectedDecreaseCarValuePercentage = 20
expectedDecreaseCarValue = function(FETYPE.OTHERASSETS.VEHICLE.prop.yearFirstRegistered,
                                    FETYPE.OTHERASSETS.VEHICLE.prop.value) {
    var expectedDecreaseCarValuePercentage = 20;
    var currentYear = new Date().getFullYear();
    var age = currentYear - yearFirstRegistered;
    var factor = 1 - (expectedDecreaseCarValuePercentage / 100);
    var lastYearValue = Math.pow(factor, age) * value;
    var impairmentThisYear = lastYearValue - (lastYearValue * factor);
    return impairmentThisYear;
}

addedToNetWorth = solvencyAfterLivingExpenses + savings + propertiesValueIncrease - expectedDecreaseCarValue
```

Note that the anonymous functions must be written inline in the actual program but are identical otherwise. They are displayed on multiple lines for readability in this text.

This is the whole DSL program and it is 150 lines, including comment headers, whitespace and nicely formatted JavaScript. For comparison, the average class length in our production code base is 130-140 lines. Usually multiple classes are needed to make a single feature or calculation. The biggest classes are 2700 lines and they are painful to maintain. Clearly, we would not be able to make the same mess with 150 lines of code.

Most rules should be self-explanatory if the reader is familiar with the English terms of the financial domain. A majority of the rules are simple arithmetic operations of no technical interest. We will therefore discuss the more complex operations and the other aspects of software development using this solution. In case of confusion we advise looking up chapter 3 or documentation on JavaScript.

The most complex function is the calculation of monthly loan expenses:

```
//Expenses
householdExpenses = FETYPE.EXPENSES.HOUSEHOLD
carExpenses = FETYPE.OTHERASSETS.VEHICLE.prop.sumExpensesFields
housingExpenses = FETYPE.OTHERASSETS.OWNEDHOUSING.prop.sumExpensesFields
installmentLoanMonthlyExpenses = function(FETYPE.LOAN.INSTALLMENT.prop.value, FETYPE.LOAN.INSTALLMENT.prop.durationMonths,
                                          FETYPE.LOAN.INSTALLMENT.prop.nominalInterestRate,
                                          FETYPE.LOAN.INSTALLMENT.prop.instalmentFee) {
    var amountThreshold = 100;
    var iterationLimit = 100;
    if (!instalmentFee) {
        instalmentFee = 0;
    }
    var calculate = function(value, duration, amount, nominalInterestRate, instalmentFee) {
        var monthlyInterestRate = nominalInterestRate / 1200;
        var month = 1;
        var finished = function(value) {
            return (value < 0 && Math.abs(value) < amountThreshold) || (value > 0 && value < amountThreshold);
        };
        var interest;
        var monthlyMortgage;
        while (month <= duration) {
            interest = value * monthlyInterestRate;
            monthlyMortgage = amount - interest - instalmentFee;
            value -= monthlyMortgage;
            if (finished(value)) {
                return amount;
            }
            if (value < 0) {
                return -1;
            }
            month++;
        }
        return 0;
    };
    var interpolation = function(value, duration, nominalInterestRate, instalmentFee) {
        var amount = 0;
        var lowest = value / duration;
        var highest = value * 0.5 * (duration / 12);
        var iteration = 0;
        var pivotAmount;
        var result;
        while (amount === 0 && iteration < iterationLimit) {
            pivotAmount = (highest + lowest) / 2;
            result = calculate(value, duration, pivotAmount, nominalInterestRate, instalmentFee);
            if (result === 0) {
                lowest = pivotAmount;
            } else if (result === -1) {
                highest = pivotAmount;
            } else {
                return result;
            }
            iteration++;
        }
    };
    return interpolation(value, durationMonths, nominalInterestRate, instalmentFee);
};
```

This is a function that contains two inner functions. This means they are functions inside a function, an important part of encapsulation in functional programming. The first one called `calculate()`, it simply calculates a loan and returns -1 if the installment is too high. The second function is called `interpolation()` and uses the first function. It iterates through a bunch of inputs until it reaches a result that is accurate enough or, the iteration limit has exceeded.

We call the interpolation at the last line of the outer function, returning the monthly expenses for a given loan. This function is called for each loan but the results are added before assigned to the result variable `installmentLoanMonthlyExpenses`.

If this loan calculation was put in production it would be easy to assure the quality. Because it is a standalone function, it could be included in the engine as an imported file. We could then run automated tests, expand functionality and refactoring without worrying about side effects in the rest of the program. This is a direct result of the functional programming.

Reactive and functional programming have many shared attributes like no global states, no singletons and focus on the relationship between data instead of the actual transformations. This should allow us to combine both techniques more easily.

The thesis certainly benefitted from this when I had to solve the more complex problems. Because the reactive system was already in place with nodes, relationships and functions, I only had to replace the functions. The code remained unchanged for fetching data, updating, calculating and visualizing.

The functional programming appeared to naturally integrate with the reactive system and its use of a language like JavaScript with first-class functions. Such an integration without side-effects can only be contributed to the reactive and functional architecture [5].

4.2 Readable business rules

Some of the business rules are quite compact. The code is not made to explain why and what is happening. It is made to explain how. The programmer or person modifying the code is still required to have a sufficient understanding of the domain.

This is no different from the current imperative code but we often implement all sorts of sanity checks and workarounds. For instance, we calculate the wealth tax with the following code:

```
wealthTaxRatePercentage = 0.7
wealthTax = function(networth, wealthTaxRatePercentage) {
    if (networth >= 1480000) {
        return networth * (wealthTaxRatePercentage / 100);
    } else {
        return 0;
    }
}
```

We can very clearly see how the calculation is done. If the value is 1 480 000 or more we will calculate the tax using the percentage constant. Otherwise, the tax is 0. If the government changes the rule from a constant limit to only a percentage of the net worth, we can easily change it. There are no structures around that needs to be maintained.

This of course goes the other way around as well. If there were 20 constants, this code would look rather messy and less maintainable. But it would still only be 100 lines of compact code that is isolated and that is always maintainable in the long run.

Understanding how the code works and how to maintain it is not everything. Intentions and the reasoning behind them are important factors in a software's lifecycle. Often the code works in a different way than what you would expect because of historical reasons. The requirements and models changes automatically with time but code and architecture does not. This is seen both in practice and in popular computer manifestos [6] [7].

Below is an example where the logic seems less intuitive for a programmer. The following code makes sense but with at least one exception:

```
yearlyCreditExpenses = installmentLoanMonthlyExpenses * 12 + creditLoanExpenses

miscellaneousExpenses = FETYPE.EXPENSES

savings = FETYPE.EXPENSES.FUND.SAVINGSAGREEMENT

fixedExpenses = miscellaneousExpenses - householdExpenses +
                housingExpenses + yearlyCreditExpenses + carExpenses - savings
```

We want to calculate fixed expenses. We add together all the expenses and then we remove savings. Savings is subtracted from the income and thus considered an expense but because the money is not transferred to someone else, it is not considered a fixed expense.

The variable “householdExpenses” is considered a fixed expense but is also removed. This might seem wrong but the reasoning is purely financial. It is because you will always need clothes, food and medicine. They are costs that are considered special and cannot be removed or changed notably.

None of this is not communicated in the code at all. It is unlikely that it would be communicated clearly in other applications implementing this but, at least it would be possible through the code. It would also be more appropriate to have such comments in a class or formal document rather than in a linear text like mine.

This is currently a weakness or trade-off in my DSL. It is compact and concise, almost too much. Like in other aspects of software, proper documentation could be the solution. Another important weakness in the current implementation is error handling. Whenever the code has a flaw, the following error is reported from Hotdrink:

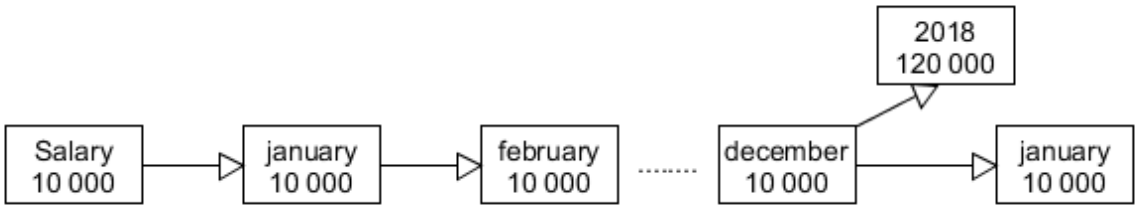
```
▶ TypeError: Cannot read property 'selectedForConstraint' of null
    at h.<anonymous> (file:///C:/arbeid/server_newname/part6_webapplivedata/lib/hotdrink.min.js:4:19722)
    at Array.map (native)
    at h.evaluate (file:///C:/arbeid/server_newname/part6_webapplivedata/lib/hotdrink.min.js:4:19688)
    at h.update (file:///C:/arbeid/server_newname/part6_webapplivedata/lib/hotdrink.min.js:4:18214)
    at h.performScheduledUpdate (file:///C:/arbeid/server_newname/part6_webapplivedata/lib/hotdrink.min.js:4:18085)
    at e.run (file:///C:/arbeid/server_newname/part6_webapplivedata/lib/hotdrink.min.js:1:8090)
    at e (file:///C:/arbeid/server_newname/part6_webapplivedata/lib/hotdrink.min.js:1:7599)
```

This is expected because Hotdrink is researchware. Researchware is ment for research and not for production. This means my proof of concept also is researchware at best because it relies on Hotdrink. However, I do suspect a variation of this problem is a more common weakness in all reactive systems due to their abstract nature.

4.3 Implementing the time dimension

Implementing the time dimension itself is quite easy. It is just another axis. This could easily be implemented by making all variables into an array. This is already a feature supported by Hotdrink but it is not documented. We could then scale the array to represent years, months or days.

We then have to connect the nodes. This is where the hard part comes in. Immediately we have multiple difficult problems. First, we need to connect the nodes together. For income, this is straight forward. We simply take the value from the current node, connect it to the next with an addition:



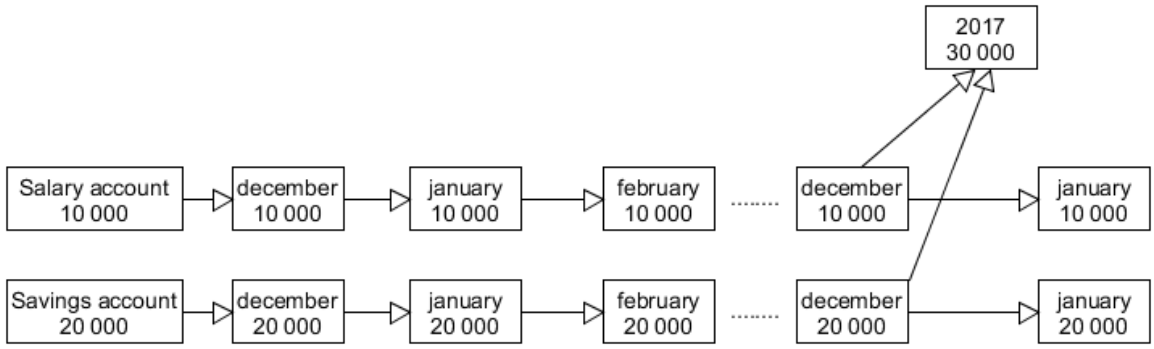
Nothing changes a node except the previous node or removing the node. Everything seems easy so far.

Then we have to change a salary's initial value and it might seem like a trivial task but it is quite cumbersome in Hotdrink. To change the value of a salary, all its nodes must change. This will generate n calculations for n months. If we calculate 10 years this is 120 calculations. This can be avoided by turning the calculations off when we change the node network.

Turning it on and off creates a higher level of concurrency because it was previously encapsulated in the reactive system. We now need to know whether the system is calculating or not. If we have more than one thread working, this can become complex very quickly with deadlocks, performance issues and more. Still, after all the apparent problems, being able to do more than one thing at once is highly valued or possibly critical.

Before moving on we will briefly touch on the concept of aggregated values. In the figure above we can see a node on the right called "2018" with 120 000 as value. How this number is calculated is a topic in itself and will not be discussed further in this thesis. For now we just assume it is calculated correctly with the data you would expect from the salaries, 12 months multiplied with a salary of 10 000 is 120 000. This is quite simple and makes sense.

However, if we now introduce the asset nodes, it is more complicated:



We can see that the nodes are no longer added. At the end of the month, your account balance is naturally not added to next month, it simply continues. We still need to summarize the state of all accounts at the end of the year. We can see a trend were different kinds of data require different kind of connections and aggregations

In the examples above we generalized to months with yearly aggregated values. In real cases we need all three levels of detail, daily, monthly and yearly. Daily is the most complex by far. We can avoid some complexity by simplifying some specifications.

Varied length of months is a good example of such simplifications. Because varied length is quite simple to understand, it is postponed until later. It simply means we must calculate with different number of days in months, e.g. 28 for February and 31 for December.

This probably seems quite simple but the consequences are huge. We now have to do a lot of work with dates. February is different in leap years. Ultimo, last day in month, is now a different day every month. Payment dates above 28 will have to be moved. We could list out many pages of such special cases but we will move on and explain why varied length of months is important.

Banks use daily accuracy when they calculate interest. This means that you should pay the same amount of interest whether it is a leap year or not. The extra cost of an extra day in a leap year is divided and subtracted from all the other days. This results in a lower daily interest than regular years because the yearly cost must be the same.

This is opposite for months, months with 31 days have higher interest costs than months with 28-30 days. For fixed rate loans this is hidden because the monthly payment is fixed. If you look closer on mortgage payment plan you will be able to see the variation in interest and mortgage. This effect sometimes results in more interest paid the month after. This is counterintuitive because the loan was reduced last month, thus there is less to calculate interest from and less expected interest cost.

This is an example from the financial domain where the calculations are too accurate for practical use, but we are still required to implement it by the government. The private customer will sign a legal document containing a payment plan. It is then important that the numbers are accurate, in accordance with the law. This is still silly because such loans are usually issued with floating interest rate which means the numbers are out of date once the rate changes. The need for accuracy is only a legal issue, it has no other use for our customers.

Implementing a dimension of time is no longer a research goal in the thesis. We will therefore evaluate it here.

I conclude that Hotdrink is a low-level API compared to our financial-economic domain with all the complexities from timescales, massive amounts of data and thousands of business rules. It should be obvious that we do not want to write all this daily, monthly and yearly logic over and over for each node. We need to map which kind of nodes are related to monthly and yearly aggregation. Implicitly that relation also needs to be defined as subtraction, addition or just continuation.

I think lacking competence and experience is the main reason why all this seems so hard. With more planning and expertise for help, implementing a dimension of time might not be so difficult after all. At least we should be able to find out whether it really works well or it is just technically possible.

It might seem that I am skeptical of the reactive approach but I am really not. There is even proof that the reactive approach is excellent for financial calculations. Microsoft's Excel spreadsheet and all its variations have millions of users and have been used for over three decades. Functional reactive programming in finances have been thoroughly tested because it is a core feature in spreadsheets.

5 Evaluation

All the results in this thesis are considered and intended to be qualitative and not quantitative. This was expected with the loosely formulated main research problem: *"Give a proof of concept for financial calculations using reactive programming"*. The terms "proof of concept" and "financial calculations" are not very precise criteria.

Most of the issues in the domain were well understood, creating a useful solution was not the difficult part. Attempting to evaluate my solution in an objective way was more difficult. The evaluation criteria are readability, maintainability, extensibility and efficiency. These are often rather qualitative and contextual measurements that can be difficult to evaluate, even with comparisons. There are more advanced techniques to measure my evaluation criteria, but there was not enough time.

This thesis contains a great deal of interdisciplinary topics and problems that are not well known. I have been unable to find any comparisons for this except my company's current "**Rådserver**" solution, see chapter 1.2. It is many thousand lines of old code and written in an old imperative language. Comparing the new declarative DSL to such a legacy program will give vague results at best, considering all the other commercial factors.

With no comparisons to establish a baseline, the evaluation of my solution and findings will be mostly qualitative. The evaluation will consist of four parts. The first two are evaluating the thesis result from both the business and the academic standpoint. Third, we will discuss the technical solution which can be used for tax calculations. Finally we will discuss the most important but subjective topic; how reactive programming can be used to implement financial-economic systems.

Because we have few numerical results and comparisons, we will mainly evaluate the thesis on how much functionality we can achieve with a minimum of effort and code. We will also discuss what my experience and conclusions from technology are.

5.1 Evaluation from the business standpoint

5.1.1 Direct business value

The direct business value is moderate, depending on which stakeholder you ask. There are at least three stakeholders from my company. First we have the architecture group, consisting of developers and architects. Second we have the product owners who want to research better solutions for finance problems. At last we have the human resources who wants me to finish within a reasonable time.

The most commercially interesting problem was the dimension of time, as explained in chapter 4.3. This problem is highly prioritized by all parts but we already know it well. It is at least very difficult or time consuming. There is so much uncertainty regarding both the solution and the size, this task has been postponed for 5-10 years.

To the product owners it has become a sort of holy grail. If we can just find some good solution and architecture to solve this problem, we can focus our efforts on it. So far, we do not have a good solution to the problem. Currently our only viable option is rewriting the original Borland Pascal code to Java.

Because of this old but urgent issue, there were hopes that I could make a proof of concept that could later become a full solution. The architecture team was skeptical of this because it is a bad recipe for a balanced and planned software system. I agreed completely with the architects, none of us had any extensive experience or theoretical depth in reactive systems. Useful results other than experience and insight into the problem domain should be considered a bonus.

Coding is supposed to only be a 50% of the work on the thesis. This means the constraints for the project were very strict and we had to limit the scope accordingly.

During the thesis we discussed quite a lot of issues in depth. They were well-known topics for my colleagues. Using reactive programming as a solution was very promising but we had reservations. A reactive system solves a lot of our issues but it forces tight coupling between the business rules and the reactive system.

It is not like functional programming where you could write one small component in a functional style and the rest of the system as imperative. With a reactive system you essentially give up control of application flow and data. Everything must be described in the declarative language. If you need to do something different or concurrent, it can be completely impossible.

Most of my colleagues have 10-30 years of experience with coding. We know the only constant is change. It is then critical to choose an architecture that enables change and is suitable for the changes that will come.

We brainstormed how reactive systems could solve our problems and we came up with a lot of important questions. Is it fast enough? Can it be optimized? Can we do concurrent operations? Can it be extended? Which framework to use? Which language to use?

Luckily we could discuss our questions with professor Jaakko Järvi at the University of Bergen, Hotdrink's creator. Our conclusion together is that reactive systems in general does not allow concurrent operations, optimization outside the declarative language or external extensions. Mainly because the reactive system demands complete control over execution and data. This means we cannot access any variable while the systems computes, constraining our options massively.

This ruled out a solution proposed in many such cases: a two-layered architecture with reactive data flow at the bottom. The reactive approach forces us to solve problems under constraints that are less suitable for financial-economic domain. It is highly likely that we would come across essential problems that cannot be solved adequately. This would normally not be an issue because we could hook in a component to fix it or "code around the problem".

Such isolated fixes are based on the control of flow in the application. E.g. "We can solve it here because it is done after task X and have not started on task Y". This is no longer possible but we still have to solve those problems.

I believe this mismatch between the wide specter of problems and few available tools to be the biggest weakness in reactive programming. Of course, lack of tools could be explained by my lack of skills and experience with the reactive approach.

I currently reject reactive systems as a solution to my company's problems because it is likely to become an organizational issue of competence. We have little experience with reactive systems and no significant experience using it for financial calculations. This has nothing to do with the technology itself but I will currently consider it unfit for our commercial needs because of all the uncertainty and the risk of failure.

5.1.2 Technical risk assessment

I believe the project has been a success as a technical risk assessment. We now know a lot more about how the reactive approach can be used to solve financial problems. We know it is very efficient for some problems but the most complex ones are still difficult. All this at the price of a cheap master's thesis.

When we are trying to solve a very difficult problem, the last thing we want is additional complexity unless it makes the actual task easier. As a company we also have to consider other factors than technology and possibilities. Examples of such factors are how many developers we have, what proficiencies they have and how much we are willing to invest in research.

These factors would be quite costly in my organization if we were to use reactive programming. The more difficult tasks are especially prone to costly development because the solution of choice could very well be wrong or flawed without us knowing. This is in accordance with our previous experiences from adopting new but simpler technology.

Using a programming language with first-class functions like JavaScript seems highly beneficial with reactive systems. A first-class function is a function that can be a value, commonly known as a function object. This allows us to use functional programming, a powerful technique that, when applied correctly, enable better and cheaper applications because of simplicity and no side effects.

First-class functions were expected to be very useful and this was confirmed during the work. An ecosystem for testing and development seems to be very cost-effective because the coding is done at a relatively low-level. This might be because of all the work with the compilator. I still think it would be very useful during development and maintenance because that is the case for all other types of development.

When deploying the application in a production environment, it would run on a server. This was not the case during development. The code was developed using HTML pages that referred to scripts to be executed by the web browser. In this approach it is easy to make a demo user interface.

The execution of JavaScript in a server environment is a mature and well-known technology. However, my company has no experience with such technologies and it is likely to give us new challenges during development with regards to testing, code reviews, etc.

This means we have no data on how our application will perform in production. Unable to know what performance we can expect in production is another source of uncertainty. Not only is performance is a critical quality for production, it could result in a complete project failure given the current limitations when optimizing a reactive system.

My technical risk assessment of reactive systems concludes they are currently not useful for our financial calculations and architectural needs. This is mainly because our domain as a whole does not really fit that well with the reactive architecture. There are very few

reactions to take care of. There are complex concepts like time which is both dynamic and fixed at the same time. There is an almost excessive amount of abstractions and aggregations. Neither of which reactive solutions fit naturally.

An exemption to this dismissive conclusion is the tax report. Because tax is only calculated yearly but with constantly changing rules, it is a special case. There are small amounts of data and dependencies but they are frequently changed. This is an ideal case to solve with reactive programming and a declarative domain-specific language [8]. I have done this in my code examples. It is easy to argue that the end users themselves without expert help, could write, test and use simple business logic.

5.2 Evaluation from the academic standpoint

5.2.1 A declarative domain-specific dialect with high conciseness and readability

From an academic perspective, the thesis is successful. We wanted to know if we could perform correct and efficient calculations coded in a simple and maintainable way. For this, a domain-specific JavaScript dialect(DSL) was created, whose sole purpose was expressiveness. The Hotdrink library handled most of the details like execution, data, initialization, etc. We could then write the business rules in clear text which was highly efficient, readable and maintainable.

The largest example in the thesis calculated a lot and was 140 lines, of which 50 were for a loan calculation. Whether this is a good or bad result can be hard to tell but, at least it is much shorter than the thousands of lines we would normally need to write in Java.

It is easier and better to evaluate how much we understand by simply reading it. When reading we usually prefer compressed information but sometimes the developers mix up the adjectives compact and concise. This usually happens when the developer is working on something difficult or interdisciplinary for a long time.

The developer begins assuming the reader has knowledge of the advanced context and its problems. The more specialized the developer is, the bigger the problem. Suddenly not only redundant information is kept out, but important business rules are as well because they were thought to be general knowledge by developer. The code will then be very compact but only concise to the few that knows the code.

I think the DSL code in proposed solution has found the correct balance in this matter. The code is not compact because there is consistent use of whitespace, indentation and descriptive variable names. Coupled with the simple mathematical syntax we all know, the code achieves a high level of readability and expressiveness.

140 lines of code is all we as programmers and business developers need to create and maintain for that functionality. This code of 5 000 characters is then transpiled to a 15 000 characters Hotdrink program which is way more complicated and difficult to read, not to mention editing it. To write all of this functionality I only spent five hours. That is about the smallest amount of time we would allocate for any kind of programming task.

However, it should be mentioned that I had to rewrite the program from scratch because all previous code was in Norwegian and without loan calculation. I also spent another full day on debugging and fixing a new error in the compiler. In light of this we could

say that only five hours is an exaggeration but, it is still a good indicator of maintainability. And maintainability seems to be very good.

5.2.2 The declarative DSL was an unexpected but useful result

I never intended to write a declarative language in the thesis because the main research question was about using reactive programming to implement financial calculations. Yet, the result was to create a compiler. This was because the Hotdrink API was too verbose to be useful for my purpose. At that point I had selected a narrow the path in the initial research question. I created a simplification of the API to learn more about the technology and its solutions. I think this was the correct decision because there was little time to complete anything else.

This path changed my result drastically from something vague and problematic to a concrete solution for a practical problem. The initial research question was very abstract but I now had a very technical answer: How well does reactive programming enable financial calculations when compiling declarative code to a Hotdrink API calls. The answer barely contains a single concept from the original thesis problem, reactive programming.

I think this course of action worked out great because it became my proposed solution to the initial research question and several other issues we have at work. The reactive programming paradigm is only a tool to solve problems, not a goal or solution in itself. I found the paradigm to be useful for my financial domain with appropriate abstractions. Data flow and execution patterns are too technical and low-level for a financial DSL.

5.2.3 Reactive programming, not weak but difficult and different

My general perception of a mismatch between the finance domain and reactive systems might be explained by my incompetence. To use the word incompetence might sound harsh but lack of experience and insufficient understanding of complex topics is in fact a problem. This is especially true for a topic like reactive programming.

For comparison, it can be much easier to predict the end result and weaknesses of other advanced technologies. Examples can be microservices or functional programming. They have well-defined boundaries and attributes, like full stack silos or no side effects. We can easily predict that communication and defining the responsibilities will be important for microservices.

For functional programming, the lack of side effects is really nice but it means that we need to explicitly handle the small configurable variations in shared calculations. Such configurations are obviously more easy to implement and maintain in stateful imperative application.

Finally we get to the point of this argument about well-defined boundaries. To me it seems that they do not exist in the same way for reactive programming. This paradigm feels like a very different kind of technology or way of thinking. In my experience, it is much harder to predict how it will work out in the end and what strength and weaknesses it will give the application. I believe this is because it does not relate to or fit my previous experience with architecture or programming.

I conclude that reactive programming has a steep learning curve after understanding the initial basic concepts. Putting the technology to good use requires more experience than other technologies because it comes with a new set of problems and solutions, quite

different from how we are used to solve them. This means that using programmers with relevant experience and choosing the correct initial approach can be much more important and critical compared to extra manpower or other resources.

5.2.4 Working with interdisciplinary economics as an informatics research topic

The initial thesis problem was a bit abstract but I have worked with my company's problems for five years. I had a good idea of what problems we were actually trying to solve, even if the technical problem was not very concrete or accurate. Understanding the underlying needs was of great help but made the theoretical part more difficult. Because my approach was more practical and pragmatic, it was less theoretic and academic until writing the report.

Unlike most master theses in Informatics, mine was heavily dependent on the financial-economic domain. It could have been a master thesis in economics using computer tools. Combining the two fields of science was difficult. Doing everything in English proved to be even more difficult than expected. I had to learn all the terminology over again because we only use Norwegian at work, except when coding. This generated a lot of extra work which meant less time for other problems.

The compiler was such a problem. Implementing compilers is an advanced topic and I could not expect to do it well the first time. I solved this by skipping a lot of basic compiler problems and rushing on to next problem to had to be solved.

I had no experience with reactive systems, frameworks or reactive programming paradigms. I had to learn all of it on the go and this has very little value for the academic results. And most importantly, the whole thesis is about solving financial problems using programming. Almost all decisions are based on the financial perspectives of concepts like time, hierarchies, abstractions and even exclusive Norwegian rules.

These are interdisciplinary topics that are hard to find any research on, especially concrete sources. The problems are too simple for economists and the technical issues too rare for computer scientists. This means the thesis is an academic contribution regardless of the result because there is so little research on this interdisciplinary topic.

I think two broad fields like finance and computer science are interesting but difficult to combine in a master thesis. The fields do not overlap well and it is hard to transfer knowledge like you would do in other more related fields, like computer science with mathematics. I think it would be hard to be successful on a master thesis like this without being a group or having years of experience.

5.3 Using reactive systems and Hotdrink for financial-economic calculations

5.3.1 Naively expecting an excellent match

Hotdrink is an implementation of a reactive system. The core of its implementation is a constraint system which can solve constraint satisfaction problems(CSP), simplex algorithms and more. These are generic problem solvers for a lot of different fields, including finances. However, the financial-economic domain we deal with is different. There are no complex

constraints or circular dependencies. This can easily be seen in the two figures of a mathematical and a financial system, see Figure 13: Tree for dependencies in a mathematical domain and Figure 14: Tree for dependencies in a financial domain.

Most dependencies in the financial structure are simple arithmetic operations. What makes this financial domain hard is the sheer amount of raw data and accomplishing the right level and type of correctness in calculations. We also want to choose a few values from datasets and calculations to present them in a minimalistic view to the users. These values can only be displayed after the calculations are done.

This why reactive systems and especially Hotdrink should be a good match with our financial domain. All the issues with control flow are handled by the framework. If we had some tool to generate the vast amount of Hotdrink code it would work great. Unfortunately, this tool does not exist so this thesis attempted to create it. I have been successful to some degree and in certain areas. The main problem was implementing a dimension of time, see chapter 4.3 “Implementing the time dimension”. Unfortunately this is yet to be solved or have a solution proposal.

We probably could figure out the problems with a dimension of time by looking at the solutions in spreadsheets and how spreadsheets are implemented. That is a larger topic and not within the scope. I suggest adding enabling features to Hotdrink or the compiler for implementing a dimension of time. These ideas are based on my work with reactive programming during the thesis and the issues that proved to be difficult but was expected to be easy.

Hotdrink is a general-purpose library for GUI and reactive implementations but many of Hotdrink’s features revolve around solving constraint satisfaction problems. These problems do not exist in our specific financial domain.

All calculations must be done in a certain sequence and a lot of the parameters are never changed. For example, we calculate the home maintenance cost using some standardized rates from the government and the house size. The rates change but the house size does not. If it does change, it would already be updated when we load in the data from the import. The reactive library was never intended to hold large amounts of static data.

When we do an analysis we always follow the time axis. When events in the future occur, we update the state and analysis. We never start with the result and move backwards like a simulation or when solving a CSP.

We have 1 exception to this and this is handled as a special case. Sometimes we want to know which starting scenario we need to have today to end up with a result scenario in the future. For this we use interpolation and vary the input until we are close enough. This is a rare corner case and should not influence the architecture because it has an optimal solution and is otherwise very costly to implement using reactive programming.

5.3.2 Evaluating a solution proposal on maintainability and extensibility

Maintainability and extensibility will be evaluated next. Before we start with maintainability, we have to briefly discuss what these two attributes means in our specific domain. This is because a software system can be changed in many different ways. Maintainability has a wide definition. A coder might talk about an enum list not being maintainable. An architect might talk about the difficulties of maintaining progress with the current architecture. To be more specific, we need to elaborate on the concept of maintainability in this thesis’ context.

Maintainability is the sum of all technical factors affected by a change in the software. When we do not want to change the system, but add to it, we are hampered or enabled by an attribute called extensibility [9]. These two definitions are less precise then we would like, but we will use them to evaluate our result.

The simplification of our two definitions is necessary because the real world is too complex. In the real world, extensibility is very closely related to maintainability. When we extend our system we also have to maintain the new extension. What if extensions are easy to implement but hard to maintain? Is extensibility high or low then? We leave this for the reader to decide but the code and its costs does not care about our definitions.

While there is a relation between extensibility and maintainability, the latter is also a lot more. The process of maintaining software involve all other disciplines of software development. Maintainability is greatly affected by planning, quality of the specification, testing and how the production environment is configured.

The way maintainability is influenced by other factors than the code can be illustrated with the concept of bottlenecks. A bottleneck is the part or process in our system with the lowest throughput. Bottlenecks can also be seen in the process of software development as well.

For example, if testing or delivery is the bottleneck or a critical obstacle, better or more maintainable code will not solve it quicker. The code however, can be written in a manner that enables high testability [10], which again can improve maintainability through test coverage.

Low testability can be compensated for by a full test suite of all the functionality in the public API. This will also improve maintainability through test coverage, but can be a lot less helpful during refactoring.

It all comes down to what context and problem domain we work in and how we apply good craftsmanship. This is a large and difficult topic which involves human factors and processes. We have ignored it in this thesis and instead focus on the application and its architecture exclusively.

5.3.3 High maintainability

This thesis resulted in a domain-specific dialect which is very similar to simple mathematics. It is easy to track the calculations. There is no state. No multiple execution paths. No technicalities. No hacks. No low level handling of data.

If an error is introduced, like a cyclic dependency, it will result in a parse error, not a runtime error. It is very easy to change or add behavior within the current limitations.

Most business logic can be implemented using the DSL but it still has higher readability then many other reactive syntaxes [1, pp. 15-29]. This is the same conclusion as in 5.1.2 “Technical risk assessment”. The new domain-specific dialect is considered to have high maintainability because it can do a lot with a minimum of code and complexity while still being very readable.

5.3.4 Low extensibility

Choosing a reactive framework as the cornerstone for an architecture will cost us some degree of control over the application and data. If we are only creating a limited reactive system, e.g. a microservice, it will only affect that system. In our case of financial

calculations, we need to get all the data. This means the reactive system has ownership over that data.

If we wanted to divide data between the reactive and some other system, we would greatly increase complexity and each system would be required to handle which parts it needs and does not. If we want to use data from the reactive system we are required to know when it is calculating, or simply accept receiving the data some time in the future.

This can be further underlined with the simple process of reading a single variable from the reactive system. This is quite a problem if it is to be read from the outside of the reactive framework and with some kind of guaranteed response time.

This is my company's greatest concern if we were to use reactive programming to do our calculations. There is no feasible extensibility outside of the reactive framework because the framework owns both data and dataflow. We are forced to do all changes and extensions inside the framework.

Being forced to solve problems in a certain way is often a part of the intention with architectural decisions. We want to force boundaries so we are not allowed to create a mess. For example, the microservice architecture applied correctly forces loose coupling and encourages high cohesion [11].

For reactive systems, these forced boundaries do not naturally follow the domain of finances. Instead they are a result of the reactive trade-offs. To avoid dealing with concurrency and execution, we leave it to the framework. The framework then requires all inputs to be abstract and atomic as either a weakly typed key-value element, or a relation between some elements.

This does not fit well with functional extensions in finances. They usually involve a service which reads some result values from a calculation when it is done and then providing a new result. That would be very difficult with the current situation, all variables would have to be fetched one by one.

We conclude that the reactive programming paradigm in its current state has high low extensibility for our specific financial calculations. This is mainly because of the need to know everything in a complete analysis, and being required to explicitly populate all data nodes and dependencies. For certain domains and smaller solutions like the tax report, the paradigm has proved to work very well.

5.3.5 Possibilities with publish-subscribe pattern or locking

During the research we identified two possibilities for extending a reactive system. They have not been investigated but will be mentioned here.

First candidate is publish-subscribe pattern, also known and instantiated as an event bus. This pattern is reactive by its nature and should be easy to implement inside any reactive implementation. Extending the reactive framework with an event bus for integration could be very successful because it is a simple and commonly used technique.

Second candidate is to solve the issue of data ownership. This can be done by introducing a global boolean state to represent whether it is calculating or not. We would then be able to extend calculations outside of the framework by working on the data when the framework is

waiting. However, this will increase complexity and possibly introduce bottlenecks, deadlocks and so forth.

It is still a possibility because the advisors' work involve many small breaks with human interactions. The system only receives input from the user which can be very sporadic.

This low-performance requirement could be exploited with such a boolean state but it would involve a potential long-term risk.

5.4 Working with reactive systems in the financial-economic domain

Most of the technical results come from making the compiler. It has to deal with both the Hotdrink API and the declarative language. This gave me insight into reactive systems and the implementations of reactive systems. The kind difficulties I had to deal with is likely to propagate when attempting to solve more complex matters. Especially when implementing features like the dimension of time, a global concept that affects almost all the business rules.

It can be difficult to understand how the code executes when working with reactive systems. The unfinished error handling in the researchware Hotdrink adds to the confusion. It seems the only effective way to locate bugs is the elimination method, executing selected parts to identifying the source of the error.

Adding a timeframe and more logic would exponentially increase the difficulty, making it less cost-effective. The difficulty is increased because of the complexity in handling multiple many-to-many relations. With everything executing inside a framework that only accepts abstractions like data and relations, we need to do all debugging inside our heads. Understanding and debugging complex cases without tools is extremely inefficient, if possible at all.

My colleagues initially suggested that we could implement some sort of separation or integration in the reactive system. For example, processing data that is not currently used in calculations. We were hoping we could use reactive programming to solve most issues in the domain and use other technologies for the rest.

Unfortunately, the suggested multi-layer solution is either impossible or at least currently not commercially feasible. This is because reactive systems in general do not allow for shared responsibility of data and dataflow.

This is an important result because it means reactive technology is a different kind of tool. To us developers, it ends up in the category of frameworks, instead of being a library or only a tool. It can still be used as a simple tool when inside a component but, as an architecture it is much more intrusive. Intrusive software can be described by its counterpart, the non-intrusive software. Non-intrusive software allows us to perform calculations while "model data sets are never modified" [12].

Even though we wanted reactive programming to be the solution to most of our problems and the basis for a new calculation platform, it currently does not fit our commercial criteria. We are neither technically nor financially prepared to invest a lot into reactive programming only to realize two years down the line, it will not benefit us in the long run. On the other hand, it could very well be the optimal long-term solution if done

correctly by experts with resources. This has certainly been the case with other commercial tools like functional programming and the agile methodology [13].

I will mention one last observation before moving on to the next subchapter. Throughout this report I mostly describe reactive programming as a technology with high potential but, not necessarily a match with financial calculations. However, as stated in the previous paragraph, I still believe reactive programming could be the optimal solution.

That is an interesting discrepancy. My perception of the possibilities is much more positive than my current "negative" impression from using it. I am not sure why this is but I think it is caused by my experience at work. As professional developers we must avoid using new techniques unless the benefit can pay for the cost and risk. This can easily become a tendency where general risk aversion prevents us from adopting new technology, simply because "it didn't seem to fix all our problems". If we do not understand or see the benefit of a technology, we are more likely to be more critical.

I believe this skepticism and the lack of reactive programming experience to be one of the most important parts of this thesis and its weak point. An experienced reactive programmer might be able to remove all my doubts by easily categorizing each of my problems and explaining the known optimal solutions. This could be very unlikely but it is also possible as this interdisciplinary domain is fairly unexplored as a whole.

5.5 Mismatch between the nature of finances and reactive systems

The nature of business requirements in our financial domain is worth mentioning. Reactive systems enforces multiple constraints. The impossibility for concurrency is the most influential and limiting constraint in our technical domain.

There is also a great deal of uncertainty in our problem domain. The only exceptions are constantly changing business rules and regular introduction of new concepts into the application. This works well today because we have a microservice architecture which isolates functionalities. Reactive systems do not allow this isolation. It basically works because all the business logic follow the same general rules for dataflow.

The whole system could become flawed if we introduced a new concept into these basic rules. Because the flaw arises in a closed virtual network of thousands of nodes, it is very difficult to locate, understand and fix if possible. Assuming we manage to create a fix, how would we know we did not break another part of the system? Our primary tool to ensure quality in existing features is automatic testing but that is quite costly with reactive networks. Considering the amount of data and all its permutations compared to other cases, this is an obstacle with high risk.

I will underline these position statements and examples with some facts.

The first fact is that economics is based on lots of numbers. In my experience these numbers are usually or always structured in a tabular or hierarchical form. We can then create different projections of the same data for different use. This is opposite of the reactive mindset. In the reactive world, we put in the data we need and define their relationships.

Now we want to add a global feature, say reduce accuracy from decimal to integers or introduce daily frequency. Reducing accuracy can be really simple in an imperative model

based application. We could simply change the model's get()-method to return integer instead of decimal, or do a conversion inside the `get()`-method to maintain compatibility with interfaces. This change can quickly become a lot of work in the reactive system because there are no models, only data and dependencies. Even more trickier would be configuring this feature on and off.

Introducing daily frequency in an imperative spreadsheet application could be done by changing the function that decides when the next event is generated. The spreadsheet would still summarize all the values correctly because it does not care as long as they are still in inside the year.

The same feature in a reactive system would be way more difficult because it could require change in all the dependencies. The reactive data structure is shaped by our approach, solution and understanding. The imperative data structure for spreadsheets will usually be similar to the physical spreadsheets, using groups of tables.

Concurrency is the other fact that is easy to explain. Financial calculations consist of multiple known subroutines whose results are later aggregated. This allows for high level of concurrent optimization because most of the subroutines are completely independent. They can be calculated without access to shared data and the results can be used both immediately and later during aggregation when the other results are done.

This is impossible with most or all reactive frameworks because they have the exclusive responsibility for all flow of control. At least the Hotdrink API does not allow for prioritization between the heavy and light calculations of source nodes. I assume this feature exist in more performance-focused implementations but then we would lose the valuable guaranteed correctness of Hotdrink.

6 Conclusion

We have investigated to what extent reactive programming is suited for the implementation of systems that support banks in providing financial-economic advice to their private customers.

To this end, we built a small such system in Hotdrink, as a proof-of-concept. This small system is a simplification of a much larger system that has been developed by Delfi Data, and which is overdue to be rewritten from scratch. Our system was evaluated on the commercial criteria readability, maintainability, extensibility, and efficiency.

We concluded that the solution's readability and maintainability was quite high because non-developers could understand and maintain most of the code. The mathematical syntax can be used for most business logic, and support for native JavaScript can solve the more complex cases.

However, extensibility is limited for financial calculations in general and especially for parallelization and encapsulation. The domains do not fit each other naturally. Finance is mainly large datasets in predefined structures, while reactive programming requires all data and relations to be explicitly defined.

These characteristics of the structures in finances, being predefined and grouped, makes it much harder to improve on the mentioned limitations and implement concepts like a dimension of time. This is because everything must be defined explicitly in reactive programming.

With regard to commercialization, the solution was too immature to be evaluated on efficiency. Our conclusion is that reactive programming alone is too restrictive for implementing systems in the financial-economic domain.

Notwithstanding the conclusions, as a by-product we developed a prototype which source-to-source compiles our new high-level domain-specific JavaScript dialect to reactive calculations. The prototype is a successful solution for the special case of tax report, but gave few results other than experience and insight.

7 Further work

We propose to further improve Hotdrink's financial strength. Hotdrink is a good tool for the job, creating a declarative domain-specific language. But as a professional I am required to consider the alternatives. Hotdrink is researchware and thus lacks the finish of a completed library.

For a commercial product to be developed cost efficiently, attributes like no error handling and code duplication are serious issues. Like the carpenter might use a hammer to hang a painting, he will certainly use a nail gun to build a house. One can assume the same cost-efficiency when using good tools also applies to software development. If we will spend thousands of hours and millions of dollars to develop software, we might as well buy or create the most cost effective tools for it.

One could simply add new features, but forking out from the Hotdrink code base might be a better idea. Creating a financial variation of Hotdrink will allow full focus on the financial domain, instead of compromises with mathematical domain.

It might be difficult but, being able to do parallel processing and create abstractions could make or break the reactive architecture for this domain.

Another option is off course to use another framework all together. This thesis confirms our initial assumption that using a language with first-class functions is very important.

Implementing a publish-subscribe pattern as an event bus inside the reactive framework could be an excellent technique for integrating the reactive system with other third parties without pausing execution.

8 References

[1] E. Bainomugisha, A. L. Carreton, T. van Cutsem, S. Mostinckx og W. de Meuter, «A Survey on Reactive Programming,» *ACM Computing Surveys (CSUR),* nr. Volume 45 Issue 4, August 2013.

[2] R. L. Glass, Frequently Forgotten Fundamental Facts about Software Engineering. IEEE Softw. 18, 3 (May 2001), 112-111. DOI=http://dx.doi.org/10.1109/MS.2001.922739, 2001.

[3] "Introduction to HotDrink," [Online]. Available: https://github.com/HotDrink/hotdrink/blob/master/docs/howto/intro.org. [Accessed 26 Mai 2017].

[4] J. W. Lloyd, Practical Advantages of Declarative Programming In Joint Conference on Declarative Programming, pp. 8-9, 1994.

[5] J. Hughes, Why Functional Programming Matters, pp.98-107, The Computer Journal, Vol.32(2), 02/01/1989.

[6] "Manifesto for Agile Software Development," [Online]. Available: http://agilemanifesto.org/. [Accessed 11th May 2017].

[7] "The Reactive Manifesto," [Online]. Available: http://www.reactivemanifesto.org/. [Accessed 11th May 2017].

[8] J. W. Lloyd, "Practical Advantages of Declarative Programming In, pp. 4," in *Joint Conference on Declarative Programming*, 1994.

[9] M. Zenger, «Programming Language Abstractions for Extensible Software Components. PhD thesis, EPFL, pp. 14,» University of Lausanne, 2003.

[10] W. Souza and R. Arakaki, "Abstract Testability Patterns," in *3rd International Workshop on Software Patterns and Quality (SPAQu'09)*, Florida, 2009.

[11] Sam Newman. 2015. Building Microservices (1st ed.), O'Reilly Media, Inc..

[12] L. Gendre, A. Olivier, P. Gosselet and F. Comte, "Non-intrusive and exact global/local techniques for," *Computational Mechanics, Springer Verlag,* pp. 233-245, 2009.

[13] C. Larman and V. R. Basili, "Iterative and incremental developments. a brief history, pp. 47-56," *Computer, vol. 36, no. 6,* June 2003.

[14] J. Freeman, J. Järvi and G. Foust, "HotDrink A Library for Web User Interfaces," in *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, Dresden, 2012.