Submitted for the Degree of MEng in Computer Science
for the academic year of 2016/17.

Run Frank in a Browser

Student Registration Number: 201349799
Student Name: Rokas Labeikis

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

I agree to this material being made available in whole or in part to benefit the education of future students.

Signature: _____ Date: _____

Supervised by:
Dr. Conor McBride

University of Strathclyde, UK
February 2017

Abstract

Frank is a strongly typed, strict functional programming language invented by Dr. Sam Lindley and Dr. Conor McBride. It is influenced by Paul Blain Levy's call-by-push-value calculus, and features a bidirectional effect type system, effect polymorphism, as well as effect handlers. This means that Frank supports type-checked side-effects which only occur where permitted. Side-effects are comparable to exceptions which suspend the evaluation of the expression where they occur and give control to a handler which interprets the command. However, when a command is complete, depending on the handler, the system could resume from the point it was suspended. Handlers are very similar to typical functions, but their argument processes can communicate in more advanced ways. The idea is to utilize this functionality in the web. Side-effects might be various events such as mouse actions, http requests, etc., and the handler would be the application in the web page.

The overarching goal of the project is to compile Frank to JavaScript and to run it in the browser. Users would thus be able to use Frank for web development purposes. This involves creating a Compiler and a Virtual Machine (abstract machine) which can support a compiled Frank structure.

Acknowledgements

I would like to express my special thanks and gratitude to my supervisor, Conor McBride for his guidance, support and patience throughout this project.

Also, I would like to thank my friends for helping me think through problems during our numerous technical discussions and contemplations.

Table of Contents

List of figures

List of tables

Chapter 1

Introduction

This chapter focuses on explaining the project motivation and objectives. Also, in the last section, the report structure is displayed.

1.1 Background

Functional languages are evolving and constantly changing in order to adapt to innovations and the ever-changing needs of software engineering. Stemming from this, the concept of Frank language was created by Dr. Conor McBride and Dr. Sam Lindley, followed by its implementation named “Frank” and the more recent release - “Frankjnr”.

“Frank”is a strongly typed, strict functional programming language designed around Plotkin and Pretnar’s effect handler abstraction, strongly influenced by (B. C. Pierce and D. N. Turner 2000), (G. D. Plotkin and J. Power 2001b), (G. D. Plotkin and J. Power 2001a), (G. D. Plotkin and J. Power 2002), (G. D. Plotkin and J. Power 2003), (G. D. Plotkin and J. Power 2009), (G. D. Plotkin and M. Pretnar 2013) papers on algebraic effects and handlers for algebraic effects. It is also influenced by Paul Blain Levy’s call-by-push-value calculus. Frank features a bidirectional effect type system, effect polymorphism, and effect handlers; Frank thus supports type-checked side-effects which only occur where permitted. Side-effects are comparable

to exceptions which suspend the evaluation of the expression where they occur and give control to a handler which interprets the command. However, when a command is complete, depending on the handler, the system could resume from the point it was suspended. Handlers are very similar to typical functions, but their argument processes can communicate in more advanced ways.

Because of the distinct features of Frank, in particular its capability to support effects and handlers, it has potential to be used in the context of web development. However, neither of Frank's implementations currently support this. Therefore, the main intention of this project is to enable the usage of Frank for web development. Using a functional language for web development could greatly ease the process of writing parsers and form validations on the web. Essentially, Frank code would be converted into JavaScript because of its support for functional programming. Frank would potentially gain all of the JavaScript functionality and could even use JavaScript third-party libraries. For example, Frank would be able to handle different http requests, such as GET or POST; it could fire events (ex. alert boxes), and handle events (ex. mouse clicks).

The most effective way to achieve this is to utilize the existing Frankjnr implementation, rewriting its Compiler and Abstract Machine (back end). The Compiler would take in parsed Frank code and output JavaScript which could be used by Abstract Machine at Run-time.

This projects requires comprehension regarding the ways Compilers and Abstract Machines work and knowledge on how to use them together. The author chose this project knowing that it would be challenging, but also recognizing the valuable learning experience.

1.2 Objectives

- Utilize Frankjnr implementation;
- Utilize Shonky data structures;
- Develop a compiler which compiles Shonky's data structures to

JavaScript data types;
- Develop an abstract machine implementation which supports the output of the compiler;
- The completed system must facilitate client-side communication of events and DOM updates between Frank code and the browser.

1.3 Summary of chapters

Chapter 2 - Related Work

The aim of this chapter is to highlight work done by others that in some fashion ties in with this project. This includes work which the author directly uses as a bouncing-off point, work that shows other attempts to solve similar problems, as well as connected projects which have been partially used in the final implementation of the project. Moreover, chapter explains chosen development tools and languages and reasoning behind them.

Chapter 3 - Problem Description and Specification

This chapter briefly overviews the main problems and challenges of the project. It briefly explains the requirement analysis stage. It, also, expands upon functional and non-functional requirements, followed by a detailed specification and explanation of the design methodology.

Chapter 4 - Initial development & experimental system

This chapter is focused on an initial experimental system. It highlights the reasoning behind it, the purpose of each component, their implementation, drawbacks and any potential improvements.

Chapter 5 - Detailed Design and Implementation of the final system

The chapter reflects upon the implementation and design of the final compiler and the abstract machine. It also explains the project structure, the developed testing framework, project connections between Shonky and Frankjnr, as well as possible improvements and alternative approaches to be consid-

ered.

Chapter 6 - Verification and Validation

This chapter explains how the outcome of the project was validated and what testing procedures were followed during and after the project.

Chapter 7 - Results and Evaluation

The principal intention of the chapter is to summarize the outcome of the project and to describing how it was evaluated.

Chapter 8 - Summary & Conclusion

This chapter summarizes the project while highlighting valuable lessons learned by the author and proposes relevant future work.

Chapter 2

Related Work

2.1 Shonky

Shonky is an untyped and impure functional programming language created by Conor McBride. The key feature of Shonky is that it supports the local handling of the computational effect, using a regular application syntax. This means that a single process can coroutine multiple other sub-processes; apart from the fact that it does not have type support, it is very similar to Frank. Its interpreter is written in Haskell, although it has potential to be ported to JavaScript or PHP to support web operations.

In the project, Shonky language data structures are used by the abstract machine; **Chapter 5** covers in detail both how and why they are used. However, Shonky interpreter is completely scrapped and is only used as a reference in solving any Abstract Machine or Compiler difficulties.

2.2 Frankjnr

Frankjnr is the newest implementation of Frank functional programming language described in "*Do be do be do*" (Sam Lindley, Conor McBride & Craig McLaughlin 2016) paper. Frankjnr has a parser for Frank syntax, thus allowing the user to utilize Frank in writing expressions. After parsing

Frank code it performs a type check and other necessary operations followed by a compilation to Shonky-supported data structures, which are then used by Shonky interpreter to run Frank.

2.3 Vole

Vole is a lightweight functional programming language implemented by Conor McBride. It has its own compiler and two matching interpreters. One of the abstract machines is written in Haskell and provides the opportunity to run compiled Vole programs on the local environment. The other machine is written in JavaScript, thus making it possible to run Vole programs on the web. There are a few working examples in the git repository of Vole. It also has some support for effects and handlers, so it utilizes the ability to communicate between compiled Vole programs and the front-end of applications, on run-time.

In terms of its relevance to this project, Vole is a useful resource because it essentially tries to solve the same problem, only for a different language. The author had to study Vole in order to understand its technical implementation, along with its usage of resources, while also enhancing his knowledge of compilers and abstract machines. Lastly, Vole is used for evaluation purposes as another benchmark comparison.

2.4 Development Tools and Languages

This section will briefly explain tools and languages used, and the reasoning behind each of them.

2.4.1 Vagrant

Vagrant is an optional tool which was used to create a separate development environment for the project. It runs on Virtual Box and is essentially a

server on one's local computer for booting up and working. Vagrant comes with up-to-date, relevant libraries out of the box. The author used it to maintain a clean environment and to avoid the installation of software and library management on the local machine, thus speeding up development.

2.4.2 Webpack

Webpack is a module builder and is available as an npm package. In this project, it is used for abstract machine development, as it adds desired flexibility to plain JavaScript, and compiles everything into a single light JavaScript file. The most important feature of webpack is its ability to create and export different modules; for example, a module could be either a function or a variable. This feature allows for an improved project structure as the user is able to keep JavaScript components in separate files, which makes it easier to develop, maintain and navigate through code.

2.4.3 JavaScript

JavaScript is a client side scripting language. In this project, the output of the compiler is generated in JavaScript, which is then used by the virtual machine, also written in JavaScript, so that both can cooperate on run time without any further compilations.

JavaScript is used because of its key features. Firstly, it has support for functional programming by letting function arguments be other functions. Secondly, it is supported by all popular browsers. JavaScript also has lots of libraries and features which support web development.

2.4.4 Haskell

Haskell is a static, implicitly-typed, and standardized functional programming language with non-strict semantics. Haskell's features include support for recursive functions, data types, pattern matching, and list comprehensions. Haskell was chosen for compiler development because of its functional

language features, such as pattern-matching, efficient recursion, and support for monadic structures. Frankjnr and Shonky are written in Haskell as well; so using Haskell provides an easier compatibility with those projects.

2.4.5 Report Markdown

This report adapted the template of a markdown developed by Tom Pollard, because of its flexible structure and features, such as its support for Pandoc markdown and latex expressions. The report is divided into separate source files, which produces a cohesive project structure, and every source file is compiled into a single pdf file using a compiler powered by *npm*.

2.4.6 Bash script

Bash script is a series of command line instructions placed into a single file. It is generally used for automation and, in this project, it is used to build automated test cases.

2.5 Conclusion

The main three related projects are Vole, Shonky and Frankjnr. Vole is employed as a working example of a solution to address a similar problem, which encouraged the author to experiment with different approaches. Shonky and Frankjnr are used directly, because Frankjnr is the newest implementation of Frank language and it uses Shonky's interpreter for executing Frank programs. The intention was to take the existing Frankjnr and replace Shonky's interpreter with a new compiler and abstract machine, which would support web development while keeping Shonky's syntax data structures.

Chapter 3

Problem Description and Specification

This section discusses the project's problem, challenges, requirements and risks involved.

3.1 Problem overview

The main aim of the project was to develop a new compiler and a corresponding abstract machine for the new implementation of the functional language Frank, called Frankjnr. The differential characteristic for the new compiler and abstract machine is the fact that they would support web development. This gives rise to multiple challenges, the initial one being the author's initial lack of knowledge of Frank. As it is a complex language, the author had to invest effort in order to comprehend it, enquiring the paper on Frank, "Do be do be do" (Sam Lindley, Conor McBride & Craig McLaughlin 2016).

Another challenge was overcoming the complexity of the existing systems. Due to the fact that this project is not a standalone, it depends largely on Shonky's syntax implementation as well as Frankjnr's handling of Frank code. The author had to take time to scrupulously research the existing systems to discern all the intricacies and interconnections, and to devise means to

efficiently replace the existing Shonky interpreter with a new compiler and abstract machine.

One of the challenges was the complex nature of compilers and abstract machines. This was the most difficult challenge to overcome due to the breadth of conceptual detail both of them contain. An incremental approach was therefore adapted, in order for the author to learn gradually, starting from less sophisticated concepts. The author thus started the project by developing an experimental system with its own language, compiler and virtual machine in order to understand the crucial concepts of a compiler and abstract machines. During the early course of development, the author regularly consulted his supervisor Conor McBride regarding various issues relating to compilers and abstract machines.

Testing and validation also proved to be a challenge. As the project is focused on developing a completely new system without any usage of frameworks, a testing framework had to be developed; one which could run test cases and validate them without any interaction on the part of the user. The testing framework was developed by utilizing Bash script, Expect script languages, along with Node, webpack, and ghci commands. It was at first created for an experimental system and then progressively altered and improved for the final system, based on continuous observation and testing. In the end, it dramatically contributed in decreasing development time by locating bugs, especially through targeted test cases.

The project was extremely broad and involved a great deal of initial research as well as in-depth planning; development time was therefore slow, but productive. However, due to time constraints, some of the intended directions of development had to be disregarded, in order to to finish the prototype in time. One of these was the Haskell-enforced type checker. The Haskell compiler could have a type checker which would prevent any bugs related to generated JavaScript programs by enforcing its types. Such bugs are currently spotted by the console window of the browser or the testing framework. A Haskell-enforced type checker could potentially be one of key future developments.

3.1.1 Project risks

- The dependence on Frank implementation (Frankjnr);
- Estimating and scheduling development time. Due to author's limited previous exposure to the subject, it proved difficult to anticipate the time and the resources required; however, regular meetings with the supervisor ensured that the project was kept on track.
- The lack of available resources;
- The author's initial lack of experience with languages used, and the analyzed concepts (compilers and virtual machines).

3.2 Requirements Analysis

Before development, requirements needed to be gathered. These were set out during a consultation with the supervisor regarding how the system should behave and what technologies it should utilize. In addition, the author attended a programming languages seminar where Frank language was discussed, thus gaining a practical perspective on the entire project. The author also read papers on relevant topics, such as "Compiling Exceptions Correctly" (Graham Hutton and Joel J. Wright 2004) in order to broaden his subject knowledge, which assisted in mapping out the direction of the project. Further research was done to ensure the appropriateness of chosen languages and technologies.

3.3 Specification

3.3.1 Functional requirements

- To create a new back end for Frankjnr implementation:
  - Develop a compiler which uses Shonky's language (Syntax file) and outputs a sensible and correct JavaScript code structure;

  - Develop an abstract machine which can run previously compiled JavaScript code in the browser;

- To facilitate client-side communication of events and DOM updates between Frank code and the browser.

3.3.2 Non-functional requirements

- To create a set of tests which should pass before each new release of the system. Test cases should be increased after every iteration to validate new features and to ensure that previous features are not broken;
- To develop a testing framework, which would test the system using the list of predetermined test cases;
- To measure performance (in comparison with the existing back end and with other kinds of generated JavaScript);
- Client-side programming should become possible (ex. complex parser of a text field).

3.3.3 Use Cases

Since the developed compiler and abstract machine support Frank language, which can potentially be used in an infinite number of ways, use cases are not relevant to this project. Generally, the user writes Frank programs; then compiles them in order to generate a JavaScript file. The user needs to include the re-compiled machine, which utilizes previously compiled code, into their project. For more detailed information on how to use the system, check the usage instructions in **Appendix 2**.

3.4 Design Methodology

The implementation of the project is fully focused on back-end development. The adopted design methodology was iterative and incremental development (IID). The main reasons behind choosing it included the difficult nature of

the project and the author's initial lack of knowledge on the subject. This particular methodology allows the developer to focus on a few features at a time, building the system incrementally, allowing for progressive learning.

Iterative and incremental development (IID) is a process which grows the system incrementally, one feature after another, during self-contained cycles of analysis, design, implementation and testing which end when the system is finished. This means that the system is improved (with more functionality added) through every Iteration.

The core benefits of this methodology include:

- Regression testing is conducted after each iteration, with only a few changes made through a single repetition; because of this, faulty elements of the software are easily identified and fixed before starting the subsequent iteration;
- The testing of a system's features is easier, because this methodology allows for targeted testing of each element within the system;
- The system could easily shift directions of development after each iteration;
- The learning curve is much flatter than usual, because the developer focuses on a selected few features at any given time.

One of the other principal decisions was to start with a different system, an experiment throughout which the author could learn the fundamental concepts related to compilers and virtual machines and later on adapt the knowledge gained during the development of the final system. The experiment was conducted over approximately four weeks and used the same methodology - iterative and incremental development.

Chapter 4

Initial development & experimental system

4.1 Introduction

Because of the complexity of the project and the lack of the author's initial knowledge of the field, the most optimal plan was to start with small, less demanding development and to expand gradually. The intricacies included:

- Frank is a complex functional language;
- Frankjnr adds another layer of complexity, since the author needs to be aware of the implementation and the compilation process;
- Dependence on Shonky's language, because the final implementation of the Compiler must be able to understand and compile Shonky's data structures;
- The complexity of the compilers and their implementation;
- The complexity of abstract machines and their implementation;

Thus, an experimental language was developed with a matching compiler and abstract machine. Both the compiler and the machine were developed while keeping in mind that their key components will be reused for the final system; therefore efficiency, reliability, and structure were all important factors. Furthermore, the experimental system was vastly influenced

by concepts described in “Compiling Exceptions Correctly” (Dexter Kozen and Carron Shankland 2004) and “Mathematics of Program Construction” (Graham Hutton and Joel J. Wright 2004).

4.2 Experimental system

The system consists of:

- Compiler - written in Haskell;
- Language - written in Haskell;
- Machine - written in JavaScript, compiled with *webpack*;
- Testing framework - written in Bash and Expect scripts, overview of the framework can be found in **Chapter 6**.

4.2.1 Language

A simple language written in Haskell, whose syntax supports a few specific operations, such as the sum of two expressions, “Throw” & “Catch”, “Set”, “Next”, “Get” and a new reference. Each of the operations were carefully selected; their implementation in the abstract machine varied significantly, which offered a broad and valuable learning experience.

Full language definition

Language is defined as Haskell data type “Expr”. In this case, it supports only different types of expressions. An experimental language is focused on Integer manipulation; it, thus, only supports integer values. Here is a definition of an Integer value.

```
| Val Int
```

Definition of the sum of two expressions.

```
| Expr :+: Expr
```

Syntax definition of a “Throw” and “Catch” commands. If the first expression of “Catch” is equal to “Throw”, meaning an exception is raised (something goes wrong), then the second expression will be evaluated; otherwise it will be left unchecked.

```
| Throw
| Catch Expr Expr
```

Syntax definition of “WithRef” command. It creates a new reference with a given value. The first string variable of the command defines the name of new reference, the second value is an expression which describes how to compute the initial value of the new reference and the third value is the context in which the reference is valid. “WithRef” stack frame is the handler for “Get” and “:=” commands. An example expression would be: WithRef “x” (Val 2) (Val 5 :+: Get “x”). From the example, we can see that “x” is a new reference with an initial value of “Val 2” and it is valid in a context Val 5 :+: Get “x”, which would return “7”.

```
| WithRef String Expr Expr
```

Syntax definition of getting the value of a defined reference.

```
| Get String
```

Syntax definition of setting the defined reference value to be equal to some new expression.

```
| String := Expr
```

Syntax definition for evaluating two expressions one by one and taking the value of the second. In most programming languages it is defined as “;”.

```
| Expr :> Expr
```

4.2.2 Compiler

The purpose of the compiler is to take in a formatted expression of the experimental language described above and to output a JavaScript compiled data structure (array of functions, called resumptions) which can be used by the abstract machine.

Below is a definition of code generation monad. It contains a constructor "MkCodeGen" and a deconstructor "codeGen". The monad takes in next available integer value and outputs a data structure which holds a list of integers, which map to the JavaScript code of the string type, the next available number after the compilation, and the result of the compilation "val". Usually, the list of definitions will start with the input "next number" and go up to just before the output "next number".

```
newtype CodeGen val = MkCodeGen {
            codeGen :: Int -> ([(Int, String)], Int, val)
        }
```

"MkCodeGen" is used to construct a definition in "genDef" function displayed below. "genDef" returns the definition number and is used by compile function to generate new function definitions.

```
genDef :: String -> CodeGen Int
genDef code = MkCodeGen $ \ next -> ([(next, code)]
                            , next + 1, next)
```

The type of "compile" function is shown below. The "compile" function takes in a valid language expression and outputs the entry point of the compilation as well as the compilation process which can be separated into chunks of generated JavaScript code.

```
compile :: Expr -> CodeGen Int
```

"Compile" function is either used to create new function definitions or to compile expressions into JavaScript, depending on the type of "Expr". For

example, if “compile” takes an initial expression of Val 2 :+: (Val 4 :+: Val 8), through pattern-matching the case for “:+:” will be executed, and through recursion it will be executed again for the right side (Val 4 :+: Val 8) as well. The definition of the “:+:” case is shown below.

```
help s (e1 :+: e2) = do
    f2 <- compile e2
    help ("{prev:" ++ s ++ ", tag:\"left\",
                    data:"++ show f2 ++"}") e1
```

Furthermore, for each value of the expression (“Val 2”, “Val 4” and “Val 8”), the function definition will be generated, because “compile” function has a case for that.

```
help s (Val n) = genDef $
        "function(s){return{stack:"++ s ++ ",
                    tag:\"num\", data:"++ show n ++"}}"
```

After the “compile” function is done executing the structure is compiled and ready to be outputted to the file.

4.2.2.1 Formating and outputting to file

Output formating is done by “jsSetup” function. It takes the name of the array, the result of the “compile” function and outputs a JavaScript formatted string.

```
jsSetup  ::  String       -- array name
         ->  CodeGen x    -- compilation process
         ->  (  String    -- JavaScript code
             ,  x         -- result
             )
```

“jsWrite” takes an output of “jsSetup” and writes up everything into a file - “generated.js”, which is ready to be used by the abstract machine.

```
jsWrite :: (String, x) -> IO()
jsWrite (code, x) = writeFile "dist/generated.js" code
```

Example usage:

```
let xpr = Val 2 :+: (Val 4 :+: Val 8)
jsWrite (jsSetup "Add" (compile xpr))
```

4.2.3 Abstract machine

The purpose of the abstract machine is to take in a compiled program and to provide semantics for the file, so that it is runnable on the browser. It gradually builds a stack from a given data structure (located in "generated.js"), where each frame of the stack has a link to another frame. The elegant aspect of this structure is that stack frames can be saved, updated, deleted and restored, thus making the machine's structure flexible.

Data Structure of compiled expression

The data structure of the compiled expression Each data structure generated by the compiler is an array called *resumptions*. Its entries are functions which take in a stack. This way, it is possible to nest them while keeping track of the stack. Below is an example of an array of resumptions ready to be used by the abstract machine; it was constructed from the expression Val 3 :+: Val 2. The semantics of this particular structure are simple - to add two numbers "2 + 3", so the expected output is "5". Comments in the code snippet below explain the meaning of the different variables in the structure. For more complex examples, see *main/example_programs* or test cases.

```
var foo = [];
foo[0] = function (s) {
    return {
        stack: s, // stack
        tag: "num", //expression type
        data: 3 // expression value
```

```
    }
};
foo[1] = function (s) {
    return {
        stack: { // stack
            prev: s, // link to previous frame
            tag: "left", // command used for adding numbers
            data: 0 // index of next operation
        },
        tag: "num", // expression type
        data: 2 // expression value
    }
};
```

4.2.3.1 Implementation

This section focuses on explaining the detailed implementation of the experimental abstract machine. It is defined as a function which takes in an array of resumptions as an argument. The array "foo", described above, will be used as a reference throughout this section.

Modes are objects which store current stack and computation. The initial definition of mode with pre-set starting values is shown below. Because the starting stack is empty, the "stack" parameter is defined as "null"; the "tag" is an expression type; if it is equal to "go", the machine must take the "data" parameter, which is an index of the last element in the array of resumptions, in order to retrieve the next mode.

```
var mode = {
    stack: null,
    tag: "go",
    data: f.length - 1
    }
```

Each resumption is a function, which takes in a stack as a parameter and re-

turns a mode. The abstract machine will finish the compilation if “mode.tag” is not equal to “go”. However, if it is equal to “go”, then mode must be reinitialized, because the current mode is not a final value. The machine therefore retrieves the next mode from the resumptions array and passes stack to it, so that the mode has access to previous stack frames.

```
    while (mode.tag === "go") {
        mode = f[mode.data](mode.stack);
```

“f” represents a resumptions array. Considering the example “foo” array, the code above would create the following mode.

```
mode = {
        stack: {
            prev: null,
            tag: "left",
            data: 0
        },
        tag: "num",
        data: 2
    }
```

The abstract machine will continue executing while reinitializing the mode’s value every time “mode.tag” changes to “go” until the “mode.tag” becomes not equal to “go” and the stack is empty. This means that the mode is a value and that the machine is done computing.

```
while (mode.tag != "go" && mode.stack != null) {
```

If the execution continues, then the behavior of the abstract machine will differ based on the mode’s “tag” parameter. In example “foo” computation, the “mode.tag” is equal to “num”. Overall, “mode.tag” could be equal to these values:

- **“num”** - all of the basic evaluations of the given expression. The machine’s further actions depend on the value of “mode.stack.tag”. In the case of the example “foo”, its value would be “left”.

  – Addition (“**left**” and “**right**” tags) - “left” tag creates a stack frame with tag “right”, thus preparing a frame for addition by placing the current value on the top of the stack and preparing to evaluate the other expression. If the top of the stack “tag” is equal to “right”, the abstract machine can add two of the top frames together, therefore creating a new “num” mode (contains the answer of the addition) and deleting the top frame of the stack. In the case of example “foo”, the “left” case would change its mode to:

```
mode = {
    stack: {
        prev: null,
        tag: "right",
        data: 2
        },
    tag: "go",
    data: 0
}
```

Then it would be reinitialized to:

```
mode = {
    stack: {
        prev: null,
        tag: "right",
        data: 2
        },
    tag: "num",
    data: 3
}
```

And because the “mode.stack.tag” is equal to “right”, the “mode.data” would be added with “mode.stack.data” to create a new mode:

```
mode = {
    stack: null,
    tag: "num",
    data: 3 + 2
}
```

This would be the final mode, because the stack would be empty and the "tag" would not equal "go"; so the machine would return "5" as the result of the computation. Other instructions manipulate the mode's data in a similar fashion, building or destroying the stack in the process.

- "**Catch**" - creates a stack frame with the tag "catcher" and places it on the top of the stack. Its data parameter is equal to the index of second expression which will be evaluated if first expressions throws an exception.
- New reference - creates a stack frame with a tag "**WithRefRight**". It is different from the other stack frames because it has a "name" parameter, which is needed to distinguish between different references. It also holds the value of the reference in its "data" parameter.
- Next ("**:>left**" and "**:>right**") - implementation is similar to addition; however, the key difference is that if the top stack frame tag is equal to the ":>right", the abstract machine will take its data without adding anything and delete the previous stack frame. These operations evaluate two expressions but return the value of the second.
- Set (":=") - very similar to the "Get" command (described below); the key difference is that it alters the value of the stack frame which has a tag "WithRefRight" with the given name of the reference. This command utilizes a linked list stack saving structure to restore the stack while saving any changes made. An exception could be thrown if the reference is undefined.

- **"throw"** - defines that something went wrong; so the abstract machine will look for a "catcher" frame in the previous stack frames, and if it

does not find it then it will output an exception.

```
mode = {
    stack: mode.stack.prev,
    tag: "throw",
    data: "Unhandled exception!"
}
```

However, if it does, then this means that the exception was handled; therefore, the abstract machine will reinitialize the mode to continue executions by taking the "catcher" values of "stack" and "data" (some expression).

```
mode = {
    stack: mode.stack.prev,
    tag: "num",
    data: mode.stack.data
}
```

- **"get"** - goes through the stack while looking for a reference, outputs either a value of the reference if it finds it, or tries to throw an exception if it does not. It also utilizes a linked stack saving and restoring structure in order to restore the stack if it does find a reference.

After the abstract machine finishes running, the "mode.tag" is not equal to "go" and the stack is empty, it will output the final mode to the console by invoking a "printer" function for the user to clearly see the results. Finally, the abstract machine outputs the final value of the execution on the separate line for testing purposes, which is used by the testing framework to compare the expected and the actual output of a test.

```
console.log(mode.data);
```

4.2.3.2 Linked Stack Saving and Restoring

The abstract machine uses the "saver" helper function in "get" and ":=" implementations. This function allows the machine to inspect the depths

of the stack and to assign new values to the existing frames of the stack by remembering changes made. To achieve this, the abstract machine saves each frame of the stack by linking frames together; each newly saved frame thus has a link to the previous saved frame. Below, the "saveStack" function is displayed, where the "m" variable represents the current stack frame and "prev" field is a link to the previous saved frame.

```
save = {
    prev: save,
    tag: m.tag,
    data: m.data,
    name: m.name
}
```

The save is a reverse linked list of stack frames; thus it is possible to restore the original stack including all the changes made by reverse-engineering the stack. After the stack is successfully restored, all of the save data must be destroyed and parameters reset to keep future saves unaffected. A current limitation is that only one instance of a stack save can exist at a given time; however, this is addressed in the final system.

4.2.3.3 Final Remarks

The abstract machine currently supports functionality for adding expressions, creating a reference, obtaining the value of a reference, setting the new value of a given reference and throwing & catching exceptions.

Room for improvement:

- Efficiency and optimization. An example would be removing the "go" tag and calling resumptions array directly;
- Documentation;
- Functionality;

All of these points are addressed in the final implementation.

4.3 Conclusion

A preliminary experiment, implementing the essence of Frank-like execution for a much simpler language, has been completed successfully. The key lessons were:

- Haskell syntax;
- Language creation and its syntax development;
- Compiler - parsing the input and generating valid JavaScript code;
- Machine - executing the compiled expression array and managing its resources, such as the mode, the stack and the saved stack.

Key things to improve include optimization & performance.

The further plan was to roll out the lessons learned creating a compiler and a virtual machine for more of the "Shonky" intermediate language that is generated by Frank compiler. This will be covered in the next chapter.

Chapter 5

Detailed Design and Implementation of the final system

The chapter focuses on implementation, design of the final compiler and the abstract machine, as well as, any topics connected to them.

5.1 Introduction

The development of the final system started on week six of the second semester, after the end of the experimental system's development. Some parts of the code were lifted from the earlier experiment and main concepts of how virtual machines and compilers should work were reused.

5.1.1 Disclaimer

The final system uses two other systems in its code base, thus not all of the code is written by the author of this project. Author's code will be clearly indicated. Two projects connected to the system are:

- **Frankjnr** - developed by Sam Lindley, Craig McLaughlin and Conor McBride. Final system is essentially new back end for Frankjnr. Thus, the whole Frankjnr project was used and new back end was placed in *Backend* folder. Parts of Frankjnr code were slightly updated to let the user choose between the old back end and the new one. Updated files were: Compile.hs and Frank.hs; updated functions are clearly indicated with comments;
- **Shonky** - developed by Conor McBride. Final compiler uses Shonky's syntax file for its supported data structures and parse functions. However, developed system does not use *semantics* module, since it is replacing it.

More detailed descriptions of the projects can be found at **Related work** section of the report or at their respective git pages.

5.2 Project's folder structure

The final system is located in *final_implementation* folder. And all of the code for new compiler and abstract machine is located in "Backend" folder.

Table's starting folder is */final_implementation*.

Table 5.1: Folder structure

Paths and Files		Summary
		Main directory for final implementation
	Frank.hs	Main Frankjnr file
	Compile.hs	Compiles Frank to Shonky data structures
/Backend		Machine and Compiler files
	Compiler.hs	Main compiler
/Backend/machine		Abstract Machine files

Paths and Files		Summary
	webpack.js	Webpack configuration file
	tester.html	HTML which includes output.js (testing purposes)
	main.js	Main machine function
/Backend/machine/components		Includes all components to construct Virtual Machine
	machine.js	Main component for Virtual Machine
	printer.js	Component responsible for result printing
/Backend/machine/dist		Generated files
	gen.js	Compiler generated code
	output.js	Webpack generated code (every JS file is packed into one)
/Backend/Shonky		Shonky project directory
	Syntax.hs	Contains needed data structures
/Backend/tests		Test framework
	main.sh	Main file (launch file)
	testcases.sh	Contains all of the test cases
/Backend/tests/test_cases		Actual programs to test
/Backend/tests/test_cases/old		Test programs lifted from Frankjnr
/Backend/tests/test_cases/new		New test programs
/Backend/benchmark		Benchmarking scripts

5.3 Compiler

The compiler is located in *final_implementation/Backend/Compiler.hs* and it is written in Haskell. It is initiated when Frank program is called with a flag "output-js". Example with program named "foo.fk":

```
frank foo.fk --output-js
```

The developed compiler uses Shonky data structures, located in *Syntax.hs* file. Compiler receives a list of program definitions from Frankjnr compiler in a form of Shonky syntax, in particular - "[Def Exp]". "Def Exp" is a "DF" data structure shown below. And it essentially means definition of function (operator).

```
DF String   -- name of operator being defined
  [[String]]    -- list of commands handled on each port
  [([Pat], Exp)]  -- list of pattern matching rules
```

The core goal of the compiler is to compile data types "Exp" to JavaScript structures (arrays of operators and resumptions). "Exp" is defined in Shonky's *Syntax.hs*.

```
data Exp
  = EV String                       -- variable
  | EI Integer                      -- integer
  | EA String                       -- atom
  | Exp :& Exp                      -- pair
  | Exp :$ [Exp]                    -- application
  | Exp :! Exp                      -- composition (;)
  | Exp :// Exp                     -- composition (o)
  | EF [[String]] [([Pat], Exp)] -- operator
  | EX [Either Char Exp]  -- string concatenation expression
```

The compilation results of each "DF" are combined into one JavaScript data structure, forming "resumptions" and "operators" arrays in the process. Top

level function for this is “operatorCompile”. It initiates chain reaction of function calls for each function definition (“DF”) and concatenates the results into one data structure. This chain reaction of function calls consist of “oneCompile”, which starts to compile single function definition and forms an “operators” array entry. Then it calls “makeOperator”, which sets the “interface” (all available commands for given operator) and “implementation” (JavaScript function definition) by calling “funCompile”. “funCompile” is responsible for forming the actual function implementation of the operator and it proceeds to initiate “linesCompile”, which forms a “try” and “catch” blocks and calls the final function “lineCompile”, who fills the lines with compiled expressions by calling “patCompile” and “expCompile” functions with data on function’s patterns and expressions. Finally, “lineCompile” forms a return statement. To see how these expressions and patterns look compiled, see “gen.js” file. A figure 5.1 shows a lifecycle of a single compilation. In the example, the code compiles the *[DF “main” [] [([],EI 1)]* definition, which evaluates to a single integer equal to 1.

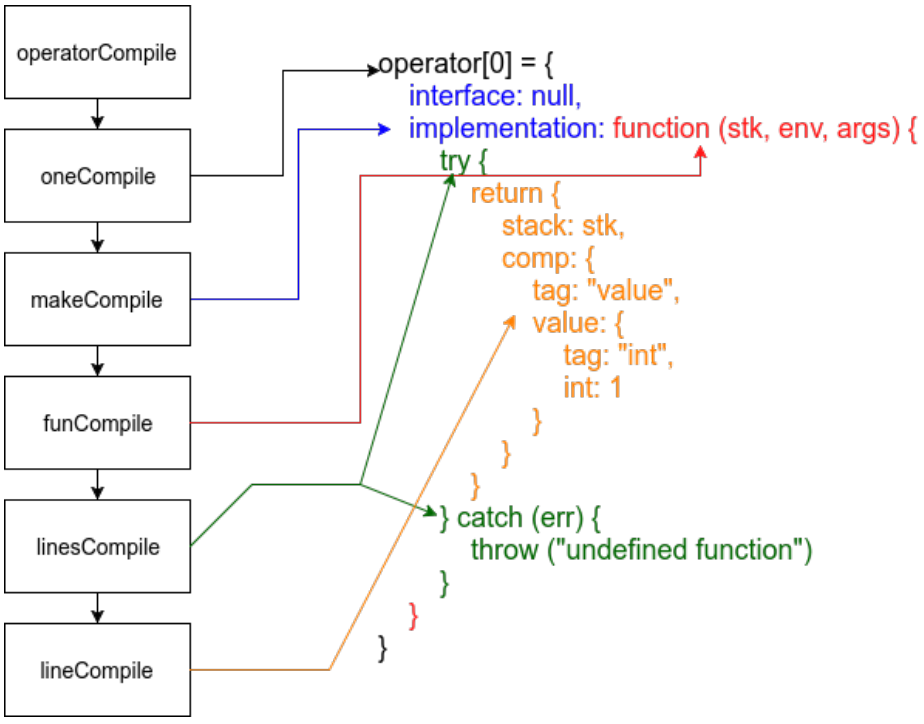

Figure 5.1: A high level representation of the compile functions

“lineCompile” will compile expressions “Exp”, however before it can do that, it must build an environment look-up table. The table is used to retrieve

defined values, shown below.

```
type EnvTable = [(String, Int)] -- environment look-up table
```

The compiler builds the lookup table by compiling list of patterns “Pat” with “patCompile” and “vpatCompile” functions. “Pat” values are retrieved from initial “DF” expression. Furthermore, there are two types of patterns. Patterns for computation - “Pat” and patterns for value - “VPat”. Patterns for computation can be thunk, command or “VPat” type. Functions “patCompile” and “vpatCompile” have been implemented using counter monad “Counter”, which is there to incrementally count each pattern and is used for indices of the environment lookup table.

In order to form the environment’s lookup table entries compiler must match patterns. The core pattern matching principle used was “match-this-or-bust” described in PhD thesis “Computer Aided Manipulation of Symbols” (F.V. McBride 1970). “patCompile” and “vpatCompile” functions, therefore, will generate series of checks for each individual pattern. If there is none failing tests, then patterns will be added to the environement, else it will throw an exception with “match failed!” message. As a consequence, the environment look-up table (“EnvTable”) will be formed with correspoding JavaScript matching tests, which will be later placed into a generated file. The code snippet below represents a case for matching “VPat” integer values. First element of the return statement is an empty environment, second is series of matching checks. In this case, the environment is empty because integers are not defined variables, so they can not be referenced anywhere else.

```
vpatCompile v (VPI x) = do -- integer value
  i <- next
  return ([],
    "if (" ++ v ++ ".tag!==\"int\") ++
        {" ++ matchFail ++"};\n" ++
    "if (" ++ v ++ ".int!==" ++ show x ++ ") ++
        {" ++ matchFail ++"};\n"
    )
```

After the environment look-up table is ready, "lineCompile" function can now use it to compile expressions - "Exp" into computations (JavaScript objects), thus forming a "return" statement in JavaScript. This is achieved in "expCompile" function, which takes in an environment look-up table, function lookup table, stack (string data structure), expression ("Exp") and outputs an JavaScript data structure encapsulated in monad "CodeGen". This monad is lifted from earlier experiment and it is used here for the same reason, to construct function definitions and track their indices in "funCompile" function. The compilation process will differ for each type of expression, because each of them have to follow different rules to be compiled correctly. For example, atoms are compiled as shown below. A JavaScript object is returned, which contains a stack and a computation with atom's values.

```
return $ "{stack:" ++ stk ++ ", comp:{tag:\"value\"," ++
           "value:{tag:\"atom\", atom:\"" ++ a ++ "\"}}}"
```

On the other hand, for example, "pair" type of expressions are slightly more complicated, because the "pair" contains two expressions, so the compiler has to evaluate them both separately. The process is similar to experimental systems addition, since the compiler makes a resumption for the second component and keeps computing the first one.

```
expCompile xis ftable stk (ecar :& ecdr) = do
  fcdr <- expFun xis ftable ecdr -- resumption
  expCompile xis ftable  -- compute first component
    ("{prev: " ++ stk ++ ", frame:{ tag:\"car\", env:env, cdr:"
     ++ show fcdr ++ "}}")
    ecar
```

Another interesting and crucial type of expression compilation is function application. Here, because the function is applied to list of arguments, compiler has to compile each of the arguments by forming a linked list data structure. And to form it the compiler utilizes a helper function, named "tailCompile", which recursively builds a linked list. To look at all of the "expCompile" cases, see *Compiler.hs* file.

5.3.1 Full compilation example

This example will show how function definition *[DF "main" [] [([],EI 1)]* is compiled. Expected output is displayed below.

```
operator[0] = {
    interface: null,
    implementation: function (stk, env, args) {
        try {
            return {
                stack: stk,
                comp: {
                    tag: "value",
                    value: {
                        tag: "int",
                        int: 1
                    }
                }
            }
        } catch (err) {
            throw ("undefined function")
        }
    }
}
```

The compilation begins with "operatorCompile", and it immediately, initiates "oneCompile" with prepared function table and single "DF" expression. In the context of this example, *"main" [] [([],EI 1)]* will be passed to "oneCompile". As a result, "oneCompile" creates "operator[0]=" and initiates "makeCompile" function with counter equal to 0, in addition to previous parameters.

"makeCompile" creates two objects: "interface" and "implementation". Furthermore, it adds all available commands to the interface object and initiates "funCompile" inside the "implementation" object. "funCompile" takes 0 (counter) and *[([],EI 1)]* as its parameters and adds

"function(stk,env,args){", before calling "linesCompile" function to form a "try" and "catch" blocks. And "linesCompile" call "lineCompile" inside "try" block, in order to form a return statement. Finally, "lineCompile" initiates "patCompile", to form an environment look-up table, followed by "expCompile" in order to compile expression *EI 1.* At this point compilation is done, because there are no more "DF"'s in the list, thus "operatorCompile" is done.

5.3.2 Helper functions

This section will briefly explain the functionality of few helper functions.

parseShonky - is used for testing purposes, it takes Shonky syntax files (ending with "uf"), reads it, parses it utilizing the parse function located in *Syntax.hs* and runs the compiler on the result. Thus, generating new "gen.js" file.

jsComplete - Takes the output of "operatorCompile". Wraps compiled list of "DF" into one structure , formats it and writes the result of the compilation to the "gen.js" file.

5.4 Abstract machine

The purpose of the abstract machine is to take in generated output of the compiler and run it on a browser, thus completing the main goal of the project. For full usage and installation instructions see **Appendix 2**.

Abstract machine modules are located in *final_implementation/Backend/machine* folder. They are written in JavaScript and are compiled to a single file *final_implementation/Backend/machine/dist/output.js* by utilizing *webpack* library.

5.4.1 JavaScript type definitions

This section explains the JavaScript types used by abstract machine to enforce the data structure of the compiled Frank programs. Firtly, "JSEnv" represents an environemnt, which is an array of "JSVal" values described below.

```
JSEnv = JSVal[]
```

"JSRun" is an operator and it always returns a mode. Operators are located in "operators" array (in "gen.js"). Mode contains a stack and current computation. Top stack frame and current computation both determine what to do next in the machine's lifecycle.

```
jstype JSRun = (JSStack, JSEnv, JSVal[]) -> JSMode
jstype JSMode = {stack: JSStack, comp: JSComp}
```

A stack could be empty or consist of a frame and a link to previous frame. The idea of these links are applied from linked list data structure, where each element of the list has a link to the next element. Linked lists are used regularly throughout the project.

```
jstype JSStack
  = null
  | {prev: JSStack, frame: JSFrame }
```

Below is a stack's frame type, which determines the operation that needs to be done when current computation is equal to "value". The operation is decided depending on the value of the "tag", so if the "tag" is equal to "car", the machine will expect that "env" and "cdr" values are defined as well.

```
jstype JSFrame
  = null
  | {tag:"car", env: JSEnv, cdr: Int }
```

```
  | {tag:"cdr", car: JSVal }
  | {tag:"fun", env: JSEnv, args: JSList Int }
  | {tag:"arg", fun: JSVal, ready: JSComp[],
     env: JSEnv, waiting: JSList Int,
     headles: Int, waitingHandles: JSList Int }
```

Linked list data structure, where it could be empty or have a head element and tailing list of elements.

```
jstype JSList x
  = null
  | {head : x, tail : JSList x}
```

A computation could either be a "value" or a "command", machine will know which one it is by checking its "tag" field and proceeding acordingly.

```
jstype JSComp
  = {tag:"value", value: JSVal}
  | {tag:"command", command: String,
     args: JSVal[], callback:JSCallBack}
```

The linked list data structure is used again to form a call back structure. This structure is used when the machine begins to search for command's handler in the stack while building a "JSCallBack" by adding checked stack frames to it. After the handler is found and command is done executing, the machine uses "JSCallBack" to rebuild the stack to its original state.

```
jstype JSCallBack
  = null
  | {frame: JSFrame, callback:JSCallBack}
```

Possible value types are displayed below, their type is determined by the "tag" value. "atom" is just an value that cannot be deconstructed any further, however in a special case when "atom" is applied to a list of arguments a

command is initiated, which is based on “atom” value. “int” represents an integer value. “pair” is a pair of two values, one is held in “car” object and the other is in “cdr”. “operator” is a top level function. “callback” holds a “callback” object which has stack frames waiting to be restored after machine finds a hadler of a command that it was looking for. “thunk” represents a suspended computation. And “local” represents local functions, which due to procedure called “Lambda Lifting” (Thomas Jonson 1985), abstract machine will turn them into a top level functions and execute.

```
jstype JSVal
  = {tag:"atom", atom: String}
  | {tag:"int", int: Int}
  | {tag:"pair", car: JSVal, cdr: JSVal}
  | {tag:"operator", operator: Int, env:JSEnv}
  | {tag:"callback", callback: JSCallBack}
  | {tag:"thunk", thunk:JSComp}
  | {tag:"local", env:JSEnv, operator: JSVal}
```

5.4.2 Implementation

The abstract machine takes contents of “gen.js” as an input, which contains two different arrays: “operators” and “resumptions”. The operators are equivalent to top level functions in Frank, the resumptions are computations waiting to be executed. In current machine’s implementation starting operator is always the first function; main method, thus, must be at the top of Frank’s file. Operators return modes which are computation which keep track of stack. Below the machine defines initial mode with initial empty stack, no arguments and no environment.

```
var mode = operators[0].implementation(null, [], []);
```

The machine will keep executing in a while loop until stack becomes empty; before halting it will return the final mode and call printer helper function to display the output. In each while cycle the machine checks the current

computation tag, it can either be a “value” or a “command” and depending on which one it is, the machine will act accordingly.

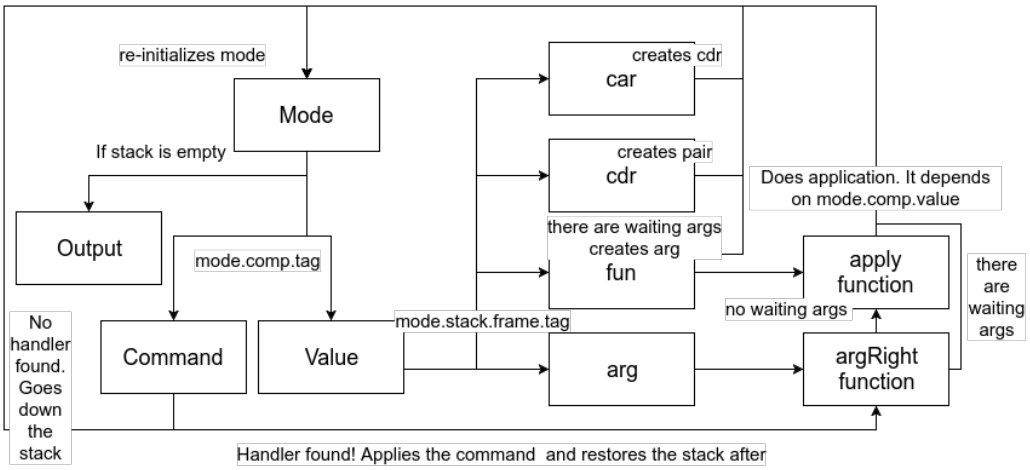

Figure 5.2: A high level representation of the virtual machine

5.4.2.1 Computation - “value”

If the “mode.comp.tag” is equal to “value” then the machine will look at the first frame of the stack to receive information on what to do next. There are four options depending on the frame tags, each of them will construct a new mode while altering the stack in the process (look at *JavaScript type definitions* section to see structure):

- **“car”** - will construct mode out of calling a resumption based on “mode.stack.frame.cdr” value. For resumption to construct a new mode it needs two values, stack and an environment. Therefore, machine will pass in the stack and the environment as well. Although, it will alter the stack by removing top stack frame and creating a new one with a “cdr” tag and a car value. This procedure is similar to experimental system’s addition operation.

```
mode = resumptions[mode.stack.frame.cdr](
    stack = { // new stack
      prev: mode.stack.prev,
```

```
      frame: {
        tag: "cdr",
        car: mode.comp.value
      },
    }, mode.stack.frame.env); // environment
```

- **"cdr"** - will reduce the stack and return a pair of *car* and *cdr* as its computation. It will take *car* value from the current stack frame and *cdr* from current computation value.

- **"fun"** - means application, so some pattern needs to be applied to a list of arguments, which is essentially list of resumptions. If the list of arguments is empty that means the machine is ready to initiate the function application by calling helper "apply" function without any arguments. If, however, the waiting argument list is not empty the machine needs to construct a new mode with frame tag equal to "arg". It creates new mode out of first argument resumption "resumptions[mode.stack.frame.args.head]" and passes all the needed information in the new stack frame:

```
frame: {
  tag: "arg",
  fun: mode.comp.value,
  env: mode.stack.frame.env,
  ready: [],
  waiting: mode.stack.frame.args.tail,
  handles: headHandles(intf),
  waitingHandles: tailHandles(intf)
},
```

"fun" field is used to keep the pattern that will be applied when the arguments "waiting" list is empty. "env" is held to keep the environment. "ready" list is used to know which arguments have been parsed, initially it is empty. "waiting" is used to keep track of list of arguments that are left unchecked. And, lastly, "handles" with "waitingHandles" are used to keep track which argument handle what commands, so if

handles list is empty it means that this argument doesn't handle any commands. It gets these values by applying helper functions, which return head and tail of the "waitingHandles" list. And the initial handle list is extracted from the operator's interface by utilizing helper function "interfaceF", which simply returns given operators interface (list of commands that it can handle).

- **"arg"** - simply adds current head of arguments to the ready list and initiates helper function "argRight" which will return new mode (its functionality in detail is described in "Helper functions" section).

5.4.2.2 Computation - "command"

Because of command type computations, Frank is different from other functional languages. When command computation happens the current execution is interrupted and the control is given to the handler, which looks for the requested command in the stack while building a callback (checked stack frames) to restore the stack when the command is found and finished executing.

In this implementation, computations with a tag equal to "command" are created in a case when an atom is applied to a list arguments. Then the abstract machine goes down the stack while looking for the command's handler. Each time it does not find it, the machine will place top frame in the callback and move down by one frame:

```
mode.comp.callback = {
  frame: mode.stack.frame, // current frame
  callback: mode.comp.callback
}
mode.stack = mode.stack.prev // new frame
```

For the machine to find a command in a frame, it must be an "arg" frame. And it must contain the command that the handler is looking for in its "handles" list (it indicates which commands does given "arg" handle).

```
if (mode.stack.frame.tag === "arg") {
  for (var i = 0; i < mode.stack.frame.handles.length; i++) {
    if (mode.stack.frame.handles[i] === mode.comp.command) {
      ...
```

When these conditions are met, the command handler is found. Therefore it will place the computation of the command on the "ready" argument list and will try to apply it by calling "argRight" helper function, which in turn will call "apply" helper function if there are no waiting resumptions.

5.4.3 Helper functions

This section will describe the functionality of two main helper functions.

argRight - takes in stack tail, function to be applied, list of arguments which are ready, environment, list of arguments which are still waiting to be checked and list of handled commands. List of handled commands is kept for the machine to know what commands does the resumption handle. Same as in the "fun" case, if the "waiting" list of resumptions (arguments) is empty then the machine is ready to apply the function, thus it will call apply function. If the "waiting" list not empty then the machine will create new "arg" frame in the same fashion as in "fun" option. The key difference is that "fun" could be instantly applied if it did not have any arguments, thus not creating any "arg" frames. However, there is a potential to move all logic to "argRight" function without having any in "fun" case, improving code reuse and optimization.

apply - Depending on a "fun" computation value, constructs a mode from which to continue execution. "fun" computation could be one of the following:

- **"int"** - Applying "int" to an argument is not really sensible, however in this case it is possible because of built-in operations (see *Built-in functions* section). This custom functionality lets machine add and subtract integer values.

- **“local”** - Means a local function. The machine is turning a local function into top level operator, concept introduced by (Thomas Jonson 1985) and it is named “Lambda Lifting”. Mode is constructed out of an “operator” variable, which contains full function definition.
- **“operator”** - Means top level function, mode is constructed from an “fun.operator” value which is an index for operators array.
- **“atom”** - Applications of atoms mean a command is initiated; mode with “command” tag and command value should be created. It, also keeps the arguments and creates a “callback” value to be able to restore the stack successfully after finding required command in the stack. Definition of new mode is displayed below.

```
stack: stk,
comp: {
  tag: "command",
  command: fun.atom,
  args: vargs,
  callback: null
}
```

- **“thunk”** - Application of thunk pattern (suspended computation). Constructs a mode while ignoring any arguments; computation expression comes from “fun.thunk”. New mode is equal to:

```
stack: stk,
comp: fun.thunk
```

- **“callback”** - This means command has been found and executed and now the machine has to restore the stack to its original position. It does this by looping through the “callback” while building the stack with callbacks frames. When it is done, machine returns a mode constructed of the restored stack and first element of arguments list.

```
    stack: stack,
    comp: args[0]
```

printer - takes the mode of finished execution, parses it to make it readable and displays it on the browser's console.

5.5 Built-in functions

Current built-in functions include "plus" and "minus", in order to make integer manipulation possible. They are defined before the top-level function compilation begins in function *operatorCompile* and then just initialized together with them, like this:

```
let fs = [(f, (h, pse)) | DF f h pse <- ds] ++ builtins
```

However their current definitions are a bit different. The compiler is not able to give them meaning because it does not know how to manipulate two integer expressions; instead compiler must pass this functionality to the abstract machine, which can interpret those expressions and apply wanted arithmetic operation. The compiler codes wanted arithmetic operation into the function definition, like this:

```
DF "minus" [] [...], EV "x" :$ [EV "y", EV "y"]
```

":$" means application, compiler tries to apply integer variable "x" to a list of integer variables "y", which initially does not make any sense. However, machine has encoded semantics for this situation. If integer, therefore, is applied to a list of arguments that could only mean one of two things, either that integer needs to be added to the first element of the list or subtracted from it. The machine determines the arithmetic operation by looking how many elements are in the argument list and applies the correct operation accordingly.

```
if (args.length === 2) { // minus
  answ = fun.int - args[0].value.int;
} else { // plus
  answ = fun.int + args[0].value.int;
}
```

Those were special cases when compiler is not able to deal with them, however other built-in functions, which do not require arithmetic operations can easily be added without machine knowing about them. Such as, pairing two elements or getting first element out of a pair.

5.6 Possible improvements

Abstract machine

- To store stack frames in chunks of frames, where only the top frame of a given chunk could potentially handle a command. When searching for a command's handler this method would allow to jump through frames, which tags are not equal to "arg", therefore improving performance and efficiency. This improvement is based on concepts described in "Proceedings of the 1st International Workshop on Type-Driven Development" (Daniel Hillerström and Sam Lindley 2016);
- Add string concatenation and built-in web feature support.

Compiler

- Current state of JavaScript type definitions are not enforced by the compiler; so compiler trusts the programmer to encode them correctly. The improvement would be to enforce types with Haskell data structures, thus minimizing the risk of bugs in production;
- To change pattern matching procedures from "match-this-or-bust" (F.V. McBride 1970) to building a tree of switches described in

"Functional Programming and Computer Architecture" (Lennart Augustsson 1985), in order to increase efficiency;

- Implement string concatenation support.

Chapter 6

Verification and Validation

This project's verification and validation were done strictly throughout testing. Development was done in two parts, firstly developing the experimental system (for learning purposes) and then the final system. Initial testing began at the early stages of the project, by testing the experimental system's abstract machine's behavior without the interaction of the compiler. At that point, it was done manually, the author typing in expressions and assessing the output; however, as development went on, this quickly became inefficient. An experimental testing framework was thus developed (February 10th, 2017), which went through pre-set test cases one by one without any human interaction whilst providing feedback for the user during the process. The author then adapted the system and applied his acquired experience to create an improved framework for the final system.

During the implementation of both systems, test cases were added regularly after each newly developed component. On the account of iterative and incremental development methodology (IDD), the author was able to write targeted test cases for each component of the system. Consequently, new bugs were identified and tracked faster, which allowed for more efficiency in software development. Test cases were initiated before every content push to git, as well as during the development, in order to check for any broken parts of the system.

Following the implementation, test framework was initiated again in order to

check whether all tests still passed; some manual testing was also performed to verify the state of the system. Furthermore, test programs from *Frankjnr* implementation were lifted and used to verify the behavior of the final system by confirming that the output of the tests were the same in both systems. See more on this in **Chapter 7**.

A possible improvement would to be to use a service similar to *Jenkins* in order to automate the tests after every push to git and once a day, regularly.

For a full list of test case expressions and programs, see **Appendix 3**.

6.1 Experimental testing framework

This section will describe the implementation of the experimental testing framework. It consists of two files utilizing two different scripts: a Bash script and an Expect script. Its purpose is to automate the testing process. Each of these scripts will be reviewed below.

6.1.1 Bash script

Bash script is the main script in the testing framework which stores test cases; then assesses them one by one. To launch the script, the user has to input *./tester.sh* in the terminal window. For a full guide on installation and usage, see **Appendix 2**.

Test cases are arrays which store free values: the expression to be tested, the name of the test and the expected output. A sample test case is displayed below:

```
declare -A test0=(
    [expr]='let xpr = Val 10'
    [name]='test_num'
    [expected]='10'
)
```

For each test case, Bash script launches Expect script and passes on to it the test case parameters - the expression to be tested and the name of the test:

```
./tests/helper.sh ${test[expr]} ${test[name]
```

After Expect script "helper.sh" finishes computing, a new program is generated. The system must then recompile the abstract machine's code to use the newly generated program. *Webpack* is used to compile JavaScript files into a single unit and to manage their structure.

```
webpack --hide-modules #recompile to output.js
```

After a successful compilation of JavaScript, Bash script has to retrieve the output of the program by acquiring the last line of the console output; it does this by utilizing the Node library and some string manipulation.

```
output=$(node ./dist/output.js); #get output
output="${output##*$'\n'}" #take only last line
```

Finally, Bash script simply compares the expected output with the actual output and displays the resulting conclusion for the user.

```
if [ "$output" = "${test[expected]}" ]; then
    echo -e "${GREEN}Test passed${NC}"
else
    echo -e "${RED}Test failed${NC}"
fi
```

6.1.2 Expect script

Expect script is used because of its ability to send and receive commands to systems which have their own terminal, which in this case is *GHCI*. The

developed script takes the name of the test and the expression to be tested. Since it is only possible to pass one array to Expect script, the script performs some array manipulation to retrieve the name and the expression passed by the Bash script.

```
set expr [lrange $argv 0 end-1]
set name [lindex $argv end 0]
```

After that, it launches the *GHCI* terminal.

```
spawn ghci
```

Then, the script waits for the character ">" before sending the command to load the "Compiler.hs" file, the experimental system's compiler.

```
expect ">"
send ":load Compiler.hs\r"
```

Finally, the Expect script sends the two following commands along with some variables taken from the array to generate the output of the compiler, and quits to resume the Bash script execution.

```
expect "Main>"
send "$b\r"

expect "Main>"
send "jsWrite (jsSetup \"$name\" (compile xpr))\r"
```

6.1.3 Possible improvements

- Speed and efficiency;
- Move test cases into separate file to improve structure;
- More useful statistics at the end of the test framework computation;

6.2 Final testing framework

Contained in *final_implementation/Backend/tests* folder. The framework is designed to launch a number of programs located in test_cases folder, and to provide feedback to the user regarding the successes and failures of the test programmes. This framework eliminates manual testing; allows for a quicker bug identification, thus speeding up the development process and easing maintenance. It is also fully written in Bash script, unlike the experimental testing framework. Some parts of the test framework were lifted from an earlier experiment; however, the key difference is that it does not use the GHC compiler directly; so it eliminates the need for Expect script (previously located in "helper.sh" file), which improves the performance of the framework. Another change is that test cases no longer take in expressions; instead, they take in paths to actual programs. This provides the ability to launch complete programs and to check their outputs.

For all test case programs, see **Appendix 3**. For usage and installation instructions, see **Appendix 2**.

6.2.1 Implementation

Similarly to the experimental system's testing framework, test cases are stored in an array. Each of the test cases are objects which store three values: the path of the test program, the name of the test and the expected output. They are looped through one by one, displaying the name of the test to the user and recompiling the test with:

```
frank ${test[path]} --output-js
```

It generates the "gen.js" in *Backend/machine/dist* folder. At this point, *Backend/machine/dist/output.js* needs to be compiled again to include the new "gen.js" file.

```
webpack --hide-modules
```

The script then simply retrieves the output with *Node* and checks whether it matches the expectation, while letting the user know if the test passed or failed (same as in the experimental testing framework). Finally, some statistics are shown on how many tests have passed or failed in total, with the corresponding percentage values.

6.2.2 Possible improvements

- Providing more statistics when all of the tests are executed;
- Timestamp tests, letting the user know both the individual test times and also the total time for all tests;
- Improve performance by adjusting the testing framework so that it does not attempt to launch programs failed to compile by Frankjnr compiler in the first place.

Chapter 7

Results and Evaluation

7.1 Outcome

7.1.1 Experimental system

During the initial development stages, an experimental system was developed with its own language, compiler, virtual machine and testing framework in order to expand the author's insight of the subject. The experimental system is able to compile and run a specific list of operations on the browser, particularly the sum of two expressions, "Throw" and "Catch", "Set", "Next", "Get" and can also create a new reference. These operations were chosen because they each have a significantly different implementation, thus offering a broader learning experience for the author. The abstract machine is written in JavaScript; it is therefore able to communicate with compiled expressions on run-time. The defining quality of the abstract machine is that it utilizes stack as its core data structure, with support for interrupts. They have connected handlers which, when initiated, start going through the stack, in order to execute a particular operation; when finished, the handlers restore the stack to its original position and give back control.

Here is an example of an expression with many of the available operations in action. Initially, variable "x" is created with value of "22"; then it is assigned a new meaning which constitutes of the sum between the previous "x" value

and the added value “11”. Moving on, “Next” (“:>”) operation is used to initiate the last expression, where the new value of “x” is added with a value “30”. The virtual machine will return “63” in the browser for this expression.

```
WithRef "x" (Val 22) (
    "x" := (Get "x" :+: Val 11)
        :> (Get "x" :+: Val 30))
```

Lastly, the computation is interrupted on “:=” and “Get” commands and control is given to the handler, in order to search through the stack and retrieve or update the reference of “x”.

7.1.2 Final system

The style of the code and a few functions of the experimental system were adapted in creating the final system. The author was successful in implementing a new abstract machine and compiler for “Shonky” data structures, which were in turn integrated into the Frankjnr project. The final system supports variables, integer values, atoms, pairs, function application, local functions, commands, thunk patterns, top level functions, and also contains built-in semantics for addition and subtraction. All of these features together offer a nearly complete development package, allowing virtually any current Frank program to be compiled by the new compiler and successfully executed by the abstract machine in a browser.

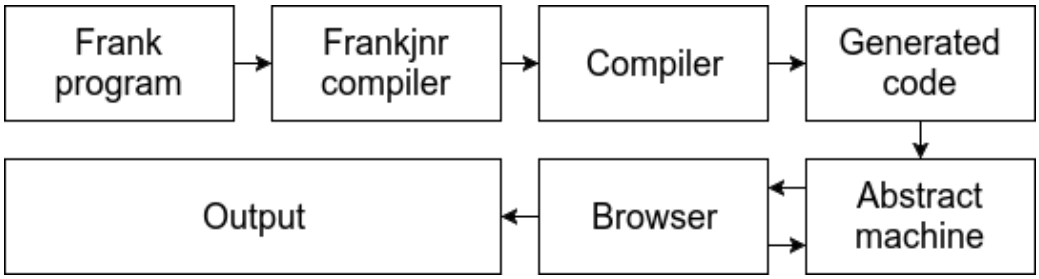

Figure 7.1: A complete life cycle

Here is a quick example of how the compiler and abstract machine would work together and how user may use it. The user is able to execute Frank

programs in four steps. The first step is to write a program in Frank with file ending in ".fk", such as:

```
main : []Int
main! = fib 5

fib : Int -> Int
fib 0 = 0
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

The second step is to compile it with a flag "output-js" to initiate the new compiler; this will generate a "gen.js" file in *Backend/machine/dist* directory with resumptions and operators arrays, which may be used by the virtual machine.

```
frank fib.fk --output-js
```

The third step is to recompile the machine's code (in *machine* folder), to include the newly generated file, with the *webpack* command. Finally, the user needs to include the "output.js" from *dist* folder, in theirs project's HTML file. In this example, the user would see "5" as the computation result in the browser.

7.1.2.1 Limitations

All of the limitations are possible improvements and they are mainly present due to time constraints.

- The compilation of Frank programs with the new compiler can be initiated only from the base final_implementation folder, because the file generation path is fixed for testing purposes;

- To be compiled successfully, a Frank program must have a "main" function and must also be the first one in the file;
- There is a lack of current web functionality support. However, because of Frank's ability to suspend a current computation, to handle commands and to restart the machine afterwards, Frank's code has a potential to query or initiate DOM updates; this functionality could be achieved in the near future;
- Some features are still in prototype stages and a few have not been implemented, such as string concatenation.

7.2 Evaluation

An experimental evaluation was chosen for this project, done in two parts. The first part was benchmarking the performance of the developed system against two other similar systems (automatic evaluation); the second part was comparing the functionality of the developed system to Frankjnr's original components.

7.2.1 Benchmark

The benchmarking framework can be found in *final_implementation/Backend/benchmark* folder. Benchmark tests can be initiated in a terminal window by typing *./evalNewBackend.sh*, *./evalVole.sh* or *./evalFrankjnr.sh*. The requirements for launching the benchmark tests are the same as the ones for the final system; however, the scripts must have valid permissions.

This benchmark was solely focused on the performance of the systems. Two types of times were taken into account: the compilation time and the execution time. The compilation time describes how fast a given program is parsed and compiled; the execution time reveals how fast a virtual machine returns the final result. A Vagrant (Virtual Box) environment with two gigabytes of allocated memory was used for running the tests. These environments are known to be considerably slower than real ones; chosen systems might

perform faster in the real world; however, in this case, systems were only compared against each other. The following systems were benchmarked:

- Final system (developed by the author of this project);
- Frankjnr original system;
- Vole.

A single test included the compilation and the execution of a given program, done 100 times, repeated 5 times for each system. The program which was executed had the same semantics for all systems. Shown below is the program used to benchmark Fraknjr and a new back-end.

```
main : []List (List Num)
main! = map {x -> cons (wrap x) nil} (cons Num (cons Num (cons Num nil)))

data Num = Num | wrap Num

map : {a -> b} -> List a -> List b
map f nil = nil
map f (cons x xs) = cons (f x) (map f xs)
```

Table 7.1: Benchmark result table

Test Nr.	Final system	Original system	Vole
Test 1	2.787 sec.	0.754 sec.	1.261 sec.
Test 2	2.847 sec.	0.882 sec.	1.293 sec.
Test 3	3.005 sec.	0.993 sec.	1.226 sec.
Test 4	2.962 sec.	1.046 sec.	1.248 sec.
Test 5	2.976 sec.	1.029 sec.	1.227 sec.
Average	2.915 sec.	0.941 sec.	1.251 sec.

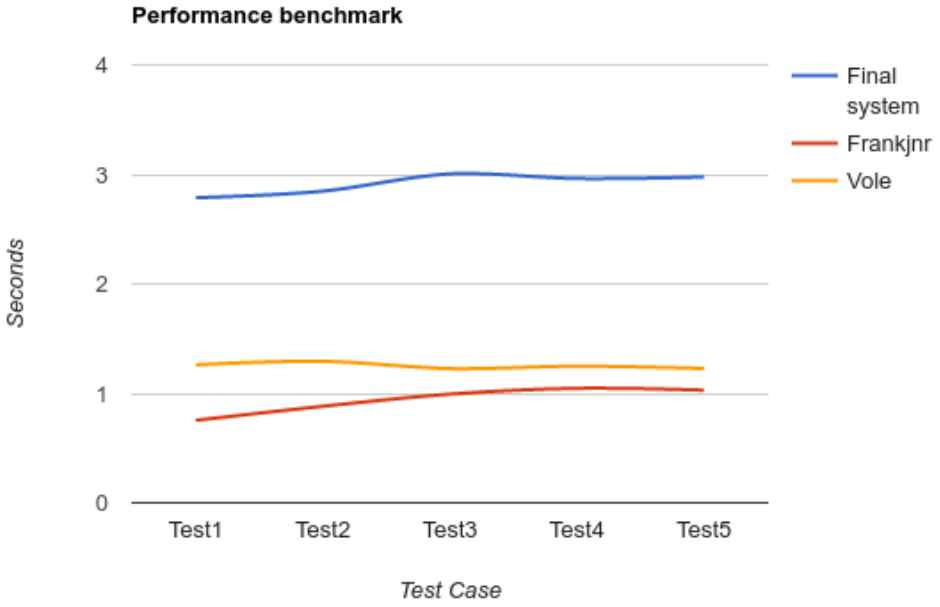

Figure 7.2: Benchmark result graph

The concerns and challenges included:

- Vole uses a different compiler with its own terminal; expect script was therefore used, which might have affected performance;
- The final system is dependent on the Frankjnr compiler;
- The final system uses a JavaScript-powered machine as opposed to others, which use Haskell abstract machines.

This evaluation gives an explicit illustration of the state of the system in terms of performance, in comparison with other similar systems. These results were anticipated, however, as performance was not one of the core goals of the project. The developed benchmark framework could be effectively reused after further development to efficiently check for changes in performance.

7.2.2 Functionality comparison

One of the main goals in creating the final system was for it to support the same functionality as Frankjnr's original components, while also adding web development features. The known main features of both systems were extracted and compared; the table below illustrates the outcomes of the comparison.

Table 7.2: Supported functionality

Feature	Final system	Original system
Integer support	Yes	Yes
String concatenation	No	Yes
Local functions	Yes	Yes
Functions	Yes	Yes
Commands	Yes	Yes
Variables	Yes	Yes
Atoms	Yes	Yes
Pairs	Yes	Yes
Application	Yes	Yes
Composition	No	Yes
Built-in functions	Yes	Yes
Parse Shonky files	Yes	No
Web development support	Yes	No

A new test program was created and executed by both systems to collect the results for each feature. If the program threw an exception, or the output was not as expected, the tested feature would not be supported by the system. In addition, the author took Frankjnr's original test cases from tests folder and initiated them against the final system, in order to see the results from a different perspective. Out of the twelve original tests, four did not pass on the final system; this was because all of them were connected to functionality, such as string concatenation, which had not been implemented at the time. For a more detailed overview of testing, see **chapter 6**.

This evaluation positively illustrates that the developed system can directly compete with the original Frankjnr back-end in terms of functionality, and that most goals of the project have been attained.

Chapter 8

Summary & Conclusion

8.1 Summary

Overall, the outcome of the project was successful. The author developed two systems with their own matching compilers and virtual machines. The initial system was developed as an experiment, to provide the author with practical knowledge on the subject, which he then applied into the development of the final system. The system is able to compile Frank code and to run it in the browser. Virtually all current Frank programs can be compiled and executed successfully by the system; the overarching goal of the project was therefore effectively obtained. However, due to time constraints, the final system is yet in prototype stages, as it does not support all of Frank's features and contains a few fixable limitations, highlighted in the **Future work** section.

During the course of the project, the author learned a variety of concepts. First of all, he expanded his knowledge on functional programming by coding the compiler with Haskell while utilizing various data types, such as monads. Secondly, the author was able to learn the infrastructure, the functionality and the implementation of compilers and abstract machines through research and participation in numerous discussions with his supervisor, Conor McBride. The author also gained a greater insight into Bash script while creating the testing frameworks for both systems and working on the project evaluation. Finally, the author learned the language of Frank

and the implementation of its defining qualities.

8.2 Future work

- Implement missing features, such as string concatenation;
- Improve the testing framework so that it show more statistics and uses timestamps;
- Expand the support for web features, such as HTTP request handling support or the handling of DOM updates;
- Improve the abstract machine's performance by storing stack frames in chunks, which would speed up searching and restoring stack, idea is based on "Proceedings of the 1st International Workshop on Type-Driven Development" (Daniel Hillerström and Sam Lindley 2016);
- Enforce JavaScript type definitions with Haskell structures, in order to decrease the risk of bugs;
- Increase compiler efficiency by updating pattern-matching procedures; building a tree of switches as described in "Functional Programming and Computer Architecture" (Lennart Augustsson 1985);
- Improve building procedures;
- Further optimize the compiler and the abstract machine;

Appendix 1: Progress log

JANUARY

January 4th:
Research:
Looked over Frankjnr, Shonky, Vole implementations.

January 5th:
Tried to install Frankjnr for a period of time, however couldn't resolve all the errors.

January 7th:
Research:
Looked at Vole with a bit more detail.

January 9th:
Tried to install Frankjnr by using Cabal dependency management tool, no success (deprecated dependencies).
Presentation & Report work:
Worked on project specification, plan and presentation.

January 10th:
Research:
Looked at Vole, in particular Machine.lhs, Compile.lhs and Vole.js.

January 11th:
Presentation & Report work:
Worked on project specification, plan and presentation.

January 16th:

Research:
Looked at Shonky stack implementation, tried to output it on the screen.

January 17th:
Machine Development:
Implemented simple linked list stack in JavaScript, just as a practice.

January 18th:
Research:
Looked again at Shonky's Abstract Machine, particularly the order the input is parsed.

January 24th:
Machine Development:
Implemented simple Abstract Machine, which can sum 2 numbers.

January 25th:
Machine Development:
Machine now works with stack larger than 2.
Updated the way it stores functions, now it uses Array data structure.
Research:
Looking into how to implement throw, catch and compiler.

January 26th:
Compiler Development:
Created basic language in Haskell. Which supports expressions and sum of expressions.
Developed monadic structure for the compiler.

January 30th:
Compiler Development:
Finished the basic layout, however it doesn't display any results yet.
Presentation & Report Work:
Setup of latex with lex2tex.

January 31th:
Machine Development:
Implemented Throw and Catch for Abstract Machine.

Presentation & Report work:
Worked on structure of the report.

FEBRUARY

February 01:
Presentation & Report work:
Worked on latex configurations.

February 02:
Machine Development:
Implemented early versions of stack saving, restoring and support for Set command.

February 04:
Presentation & Report work:
Report work - Introduction sections.

February 06:
Compiler Development:
Tried to implement Show instance for CodeGen function.

February 07:
Research:
Looked at the lifecycle and expected behaviour of the Compiler.
Compiler Development:
Implemented Compiler Catch and Throw.

February 08:
Compiler Development:
Implemented Get, Next, WithRef commands.
Machine Development:
Implemented support for Next commands.
Added new examples of working programs.

February 09:
Machine development:
Implemented early version of stack saving and restoring.
Implemented support for Set commands.

Added new working examples.
Reworked Next command support, should work as expected.

February 10:
Test Framework development:
Implemented test framework by utilizing Bash and Expect scripts.
Machine development:
Divided code into separate classes and files.
Added printer class, which just outputs the current stack to the console.

February 13:
Started to rework Abstract Machine, to closer match the implementation required.
Machine development:
Reworked Catch and Throw command support.
Compiler development:
Small efficiency adjustments.
Test Framework development:
Added more test cases.
General bug fixes.

February 14: *Test Framework development:*
Fixed major bug with string parsing.
Added more test cases.
Machine development:
Completely reworked support for Get, Set and WithRef commands.
General bug fixes.

February 15:
Progress report.

February 16:
Progress report.

February 24:
Final Compiler development:
Outputting generated code to js file.
Researched top level functions.

Project
Updated project structure.

March

March 05:
Report
Chapter 4 almost done.
Project
Improvements all around: bug fixes, documentation updates.

March 07:
Report
Related work section
Introduction chapter summarization
Final Compiler development:
Attempt to implement operators array
Looking into how to compile operator variables

March 08:
Report
Background section
Final Compiler development:
Various fixes
operatorCompile function adjustment
File Writer adjustment

March 09:
Report
Related work section
Final Compiler development:
Adjusted file generation
Fixed operatorCompile function
Final Abstract Machine development:
Initialized project structure with webpack
Initial implementation of final Machine (support for CAR and CDR operations)

March 10:
Final Compiler development:
Adjusted type definitions
Added one layer of structure to Computations
Final Abstract Machine development:
Fixed bugs regarding CAR and CDR
Optimization- "go" tag is not needed, creating modes directly
CDR now returns pair

March 11:
Final Compiler development:
Added needed functionality to compiler to support 'application' operations
Final Abstract Machine development:
Added functionality for FUN and ARG, still need to figure out how to apply a function to ready list

March 12:
Final Compiler development:
Added parser functions, now the compiler is able to covert Shonky language to its syntax
Final Testing Framework development:
Created initial structure of the framework as well as sample test case
Report
Problem overview section

March 13:
Final Compiler development:
Commands development
Final Abstract Machine development:
Commands development
Report
Specification section

March 14:
Final Compiler development:
Bug fixes
Final Abstract Machine development:

Bug fixes, command support implemented
Report
Requirement analysis section
Project structure
Frankjnr now uses new back end

March 15:
Project structure updates
Report work

March 16:
Local functions development

March 17:
Test framework development
Compiler and Machine - added Built in function support
Machine Print function development

March 18:
Report work - Final implementation section and appendixes

March 19:
Report:
Abstract machine section

March 20:
Report:
Testing and validation sections

March 21:
Benchmarking and evaluation sections

March 22:
Evaluation and conclusion

March 23:
User guide and report work

March 24:
Report work

March 25:
Report work

March 26:
Report work

Appendix 2: Usage & installation instructions

Requirements

Execute *./make.sh* file to install all dependencies, or do it manually:

- Node (https://nodejs.org/en/)

  ```
  sudo apt-get update
  sudo apt-get install nodejs
  ```

- npm (https://www.npmjs.com/)

  ```
  sudo apt-get install npm
  ```

- webpack (https://webpack.github.io/)

  ```
  sudo npm install webpack -g
  ```

- ghc (https://www.haskell.org/ghc/)
- Frank - Navigate to *final_implementation* folder and install Frank by typing:

  ```
  stack setup
  stack install
  ```

You might need to edit your machine's environment path, to include Frank.

- Expect script

  ```
  apt-get install expect
  ```

Final System

Compiling and executing programs

Navigate to *final_implementation* and compile any frank program from current directory or sub directories with flag "output-js". Example:

```
frank examples/foo.fk --output-js
```

Your current directory must be **final_implementation**!

This will generate "gen.js" in *final_implementation/Backend/machine/dist* directory. Now you have to recompile the machine's code. You can do this by navigating to */final_implementation/Backend/machine* and executing a command:

```
webpack
```

You can add flag "-w" to watch for changes.

"output.js" will be generated in the same folder as "gen.js" was (*dist*). Place it in your HTML file to initiate execution and see results of the compiled program.

Tests

Script initiates test cases and outputs the result in the terminal window.

Usage

Folder: *final_implementation/Backend/tests*.

To run tests execute command:

```
./tester.sh
```

Troubleshooting

If *tester.sh* have issues with permissions, do:

```
chmod +x tester.sh
chmod +x testcases.sh
```

Benchmark

Navigate to *final_implementation/Backend/benchamrk* directory and initiate any of the benchmarks by executing the Bash script. Example:

```
./evalFrankjnr.sh
```

Experimental System

Usage

Initiate the compiler in the *simple_implementation* directory:

```
ghci Compiler.hs
```

Define and compile valid expression, providing the name of the expression to "jsSetup" function in the process. Example:

```
let xpr = WithRef "x" (Val 2) (Val 5 :+: (Get "x"))
jsWrite (jsSetup "example_program" (compile xpr))
```

This will generate "generated.js" file in *simple_implementation/dist" directory. Now you have to recompile the abstract machine for it to include newly generated file. You can do that by executing* webpack* command in the *simple_implementation* folder. Finally, "output.js" will be generated in *simple_implementation/dist" directory, which you can include in your project in order to see the results of the compilation.

Tests

In the *simple_implementation/tests* directory execute:

```
./tester.sh
```

Troubleshooting

If files have issues with permissions:

```
chmod +x tester.sh
chmod +x helper.sh
```

Report

PDF and HTML versions of the report are located in the *report/output* folder.

Recompile with:

```
npm run watch
```

Follow usage & installation instructions described in the readme.md file, located in *report* folder.

Appendix 3: Test cases

Experimental system

This is a full list of experimental system's test cases. "expr" - is expression to be tested, "name" - the name of the test and "expected" is expected output.

```
declare -A test0=( #value
    [expr]='let xpr = Val 10'
    [name]='test_num'
    [expected]='10'
)

declare -A test1=( #addition
    [expr]='let xpr = Val 2 :+: (Val 4 :+: Val 8)'
    [name]='test_sum'
    [expected]='14'
)

declare -A test2=( # Throw
    [expr]='let xpr = Val 2 :+: (Val 4 :+: Throw)'
    [name]='test_throw'
    [expected]='Unhandled exception!'
)

declare -A test3=( # Catch
    [expr]='let xpr = Catch (Val 2 :+:
```

```
            (Val 4 :+: Throw)) (Val 2)'
    [name]='test_catch'
    [expected]='2'
)

declare -A test4=( # Catch with previous stack
    [expr]='let xpr = ((Val 5) :+: (Catch (Val 2 :+:
                        (Val 4 :+: Throw)) (Val 2)))'
    [name]='test_catch_stack'
    [expected]='7'
)

declare -A test5=( # Simple WithRef, value is not used
    [expr]='let xpr = WithRef "x" (Val 2)
                            (Val 5 :+: Val 3)'
    [name]='test_simple_withref'
    [expected]='8'
)

declare -A test6=( #undifined variable
    [expr]='let xpr = Get "x" :+: Val 2'
    [name]='test_get_false'
    [expected]='Exception: Undifined expression: x'
)

declare -A test7=( #adding defined variable
    [expr]='let xpr = WithRef "x" (Val 2)
                        (Val 5 :+: (Get "x"))'
    [name]='test_withref_get'
    [expected]='7'
)

declare -A test8=( # composition and variable defintion
    [expr]='let xpr = WithRef "x" (Val 2)
```

```
                ((("x" := (Get "x" :+: Val 11))
                 :> Get "x")'
    [name]='test_withref_get_set'
    [expected]='13'
)

declare -A test9=( # same as test 8 but plus addition
    [expr]='let xpr = WithRef "x" (Val 22)
                        ("x" := (Get "x" :+: Val 11)
                            :> (Get "x" :+: Val 30))'
    [name]='test_withref_get_set_next'
    [expected]='63'
)

declare -A test10=( # composition test
    [expr]='let xpr = Val 2 :> Val 5 :+: Val 8
                    :> Val 1000 :> Val 20 :+: Val 3'
    [name]='test_next'
    [expected]='23'
)
```

Final system

Below is a list of all test programs for the final system:

New test programs

Test program 1:

path: "Backend/tests/test_cases/new/add.fk".
Description: Peano number addition.
Expected result: [suc[suc[zero]]].

```
main : {Nat}
main! = add (suc zero) (suc zero)

data Nat = zero | suc Nat

add : {Nat -> Nat -> Nat}
add zero b = b
add (suc a) b = suc (add a b)
```

Test program 2:

path: "Backend/tests/test_cases/new/command.fk".
Description: command test.
Expected result: [pr[zero][suc[zero]]].
Test program is too long to show, check provided path for the code.

Test program 3:

path: "Backend/tests/test_cases/new/int.fk".
Description: integer value test.
Expected result: 1.

```
main : {Int}
main! = 1
```

Test program 4:

path: "Backend/tests/test_cases/new/intAdd.fk".
Description: integer addition test.
Expected result: 20.

```
main : {Int}
main! = 10 + 10
```

Test program 5:

path: “Backend/tests/test_cases/new/intMinus.fk”.
Description: integer minus test.
Expected result: 10.

```
main : {Int}
main! = 20 - 10
```

Test program 6:

path: “Backend/tests/test_cases/new/intCommand.fk”.
Description: integer values with commands.
Expected result: [pr01].
Test program is too long to show, check provided path for the code.

Test program 7:

path: “Backend/tests/test_cases/new/intList.fk”.
Description: list with integer values.
Expected result: [cons1[cons2[cons3[nil]]]].

```
main : {List Int}
main! = [1, 2, 3]
```

Test program 8:

path: “Backend/tests/test_cases/new/intLocal.fk”.
Description: local functions with integer values.
Expected result: [pr[cons0[cons1[cons2[nil]]]][cons0[cons1[cons2[nil]]]]].

Test program is too long to show, check provided path for the code.

Test program 9:

path: “Backend/tests/test_cases/new/intOperator.fk”.
Description: operator call with integer value.
Expected result: 3.

```
main : {Int}
main! = plusOne(2)

plusOne : {Int -> Int}
plusOne x = x + 1
```

Test program 10:

path: "Backend/tests/test_cases/new/lists.fk".
Description: list test.
Expected result: [cons[zero][cons[suc[zero]][cons[zero][nil]]]].

```
main : {List Nat}
main! = [zero, suc zero, zero]

data Nat = zero
         | suc Nat
```

Test program 11:

path: "Backend/tests/test_cases/new/local.fk".
Description: local function test.
Expected result:
[pr[cons[zero][cons[suc[zero]][cons[suc[suc[zero]]][nil]]]]
[cons[zero][cons[suc[zero]][cons[suc[suc[zero]]][nil]]]]].

Test program is too long to show, check provided path for the code.

Test program 12:

path: "Backend/tests/test_cases/new/operator.fk".
Description: list test.
Expected result: [suc[zero]].

```
main : {Nat}
main! = plusOne(zero)
```

```
plusOne : {Nat -> Nat}
plusOne x = suc x

data Nat = zero
         | suc Nat
```

Old test programs

These programs are all lifted from "Frankjnr" implementation tests folder.

Test program 13:

path: "Backend/tests/test_cases/old/app.fk".
Description: application test.
Expected result: 42.

```
main : {Int}
main! = app {f -> f 42} {x -> x}

app : {{X -> Y} -> X -> Y}
app f x = f x
```

Test program 14:

path: "Backend/tests/test_cases/old/evalState.fk".
Description: hello world.
Expected result: "Hello World!".

```
main : []String
main! = evalState "Hello" (put (append get! " World!"); get!)

append : List X -> List X -> List X
```

```
append nil ys = ys
append (cons x xs) ys = cons x (append xs ys)

interface State X = get : X
                  | put : X -> Unit

evalState : X -> <State X>Y    -> Y
evalState   x    <put x' -> k>  = evalState x' (k unit)
evalState   x    <get    -> k>  = evalState x (k x)
evalState   x         y         = y
```

Test program 15:

path: "Backend/tests/test_cases/old/fact.fk".
Description: factorial.
Expected result: 120.

```
main : []Int
main! = fact 5

mult : Int -> Int -> Int
mult 0 y = 0
mult x y = y + mult (x-1) y

fact : Int -> Int
fact 0 = 1
fact n = mult n (fact (n - 1))
```

Test program 16:

path: "Backend/tests/test_cases/old/fib.fk".
Description: Fibonacci generation and negative integer test.
Expected result: 5.

```
main : []Int
main! = fib 5

fib : Int -> Int
fib 0 = 0
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)

minusTwoOnZero : Int -> Int
minusTwoOnZero 0 = -2
minusTwoOnZero n = 0
```

Test program 17:

path: "Backend/tests/test_cases/old/flex-ab-eq.fk".
Description: Regression for unifying effect-parametric datatype with flexible.
Expected result: [unit].

```
main : {Unit}
main! = boo foo!

interface Eff X = bang : Unit

data Bar = bar {Unit}

foo : [Eff Bar]Unit
foo! = unit

boo : <Eff S>Unit -> Unit
boo   <bang -> k> = boo k!
boo   unit        = unit
```

Test program 18:

path: “Backend/tests/test_cases/old/listMap.fk”.
Description: map a pure (addition) function over a list.
Expected result:
[cons[cons2[nil]][cons[cons3[nil]][cons[cons4[nil]][nil]]]].

```
main : []List (List Int)
main! = map {x -> cons (x+1) nil} (cons 1 (cons 2 (cons 3 nil)))

interface State X = get : X
                  | put : X -> Unit

map : {a -> b} -> List a -> List b
map f nil = nil
map f (cons x xs) = cons (f x) (map f xs)
```

Test program 19:

path: “Backend/tests/test_cases/old/paper.fk”.
Description: examples from the paper
Expected result: “do be”.
Test program is too long to show, check provided path for the code.

Test program 20:

path: “Backend/tests/test_cases/old/r3.fk”.
Description: compiler Nontermination Issue No. 3.
Expected result: 1.

```
main : []Int
main! = iffy True {1} {2}

data Bool = True | False

iffy : Bool -> {X} -> {X} -> X
```

```
iffy True t _  = t!
iffy False _ f = f!
```

Test program 21:

path: “Backend/tests/test_cases/old/r4.fk”.
Description: problem with unifying abilities identified by Jack Williams (No. 4).
Expected result: 42.

```
main : [Console]Int
main! = apply foo!

data Bar = One {Int}

apply : Bar [Console] -> [Console]Int
apply (One f) = f!

foo : {Bar [Console]}
foo! = One {42}
```

Test program 22:

path: “Backend/tests/test_cases/old/r5.fk”.
Description: Suspended computation datatype argument.
Expected result: [unit].

```
main : []Unit
main! = foo (just {unit})

data Maybe X = just X | nothing

foo : Maybe {Unit} -> Unit
foo (just x) = x!
```

Test program 23:

path: “Backend/tests/test_cases/old/r7.fk”.
Description: issue with recursive call in suspended comp..
Expected result: [unit].

```
main : []Unit
main! = unit

interface Receive X = receive : X

on : X -> {X -> Y} -> Y
on x f = f x

receivePassthrough : <Receive String>X -> [Receive String]X
receivePassthrough x              = x
receivePassthrough <receive -> r> =
  on receive! { s -> receivePassthrough (r s) }
```

Test program 24:

path: “Backend/tests/test_cases/old/str.fk”.
Description: pattern matching list of strings.
Expected result: 1.

```
main : []Int
main! = foo (cons "abcd" nil)

foo : List String -> Int
foo (cons "ab" (cons "cd" nil)) = 0
foo (cons "abcd" nil) = 1
foo _ = 2
```

Appendix 4: Relevant README files

Frankjnr

An implementation of the Frank programming language described in the paper "Do be do be do" by Sam Lindley, Conor McBride, and Craig McLaughlin, to appear at POPL 2017; preprint: https://arxiv.org/abs/1611.09259

Installation procedure

The easiest way to install `frank` is to use stack https://www.haskellstack.org), https://github.com/commercialhaskell/stack):

```
stack setup
```

The above command will setup a sandboxed GHC system with the required dependencies for the project.

```
stack install
```

The above command builds the project locally (`./.stack-work/...`) and then installs the executable `frank` to the local bin path (executing `stack path --local-bin` will display the path).

Running a Frank program

To run a `frank` program `foo.fk`:

```
frank foo.fk
```

By default the entry point is `main`. Alternative entry points can be selected using the `--entry-point` option.

Some example `frank` programs can be found in `examples`. They should each be invoked with `--entry-point tXX` for an appropriate number `XX`. See the source code for details.

Optionally a https://github.com/pigworker/shonky file can be output with the `--output-shonky` option.

Limitations with respect to the paper

- Only top-level mutually recursive computation bindings are supported
- Coverage checking is not implemented

Report template

Template for writing a PhD thesis in Markdown

This repository provides a framework for writing a PhD thesis in Markdown. I used the template for my PhD submission to University College London (UCL), but it should be straightforward to adapt suit other universities too.

Citing the template

If you have used this template in your work, please cite the following publication:

> Tom Pollard et al. (2016). Template for writing a PhD thesis in Markdown. Zenodo. http://dx.doi.org/10.5281/zenodo.58490

Why write my thesis in Markdown?

Markdown is a super-friendly plain text format that can be easily converted to a bunch of other formats like PDF, Word and Latex. You'll enjoy working in Markdown because:

- it is a clean, plain-text format…
- …but you can use Latex when you need it (for example, in laying out mathematical formula).
- it doesn't suffer from the freezes and crashes that some of us experience when working with large, image-heavy Word documents.
- it automatically handles the table of contents, bibliography etc with Pandoc.
- comments, drafts of text, etc can be added to the document by wrapping them in <!– –>
- it works well with Git, so keeping backups is straightforward. Just commit the changes and then push them to your repository.
- there is no lock-in. If you decide that Markdown isn't for you, then just output to Word, or whatever, and continue working in the new format.

Are there any reasons not to use Markdown?

There are some minor annoyances:

- if you haven't worked with Markdown before then you'll find yourself referring to the style-guide fairly often at first.
- it isn't possible to add a short caption to tables ~~and figures~~ (figures are now fixed). This means that /listoftables includes the long-caption, which probably isn't what you want. If you want to include the list of tables, then you'll need to write it manually.
- the style documents in this framework could be improved. The PDF and HTML outputs are acceptable, but ~~HTML and~~ Word needs work if you plan to output to this format.
- … if there are more, please add them here.

How is the template organised?

- README.md => these instructions.
- License.md => terms of reuse (MIT license).

- Makefile => contains instructions for using Pandoc to produce the final thesis.
- output/ => directory to hold the final version.
- source/ => directory to hold the thesis content. Includes the references.bib file.
- source/figures/ => directory to hold the figures.
- style/ => directory to hold the style documents.

How do I get started?

1. Install the following software:

    - A text editor, like Sublime, which is what you'll use write the thesis.
    - A LaTex distribution.
    - Pandoc, for converting the Markdown to the output format of your choice. You may also need to install Pandoc cite-proc to create the bibliography.
    - Install shortcaption module for Pandoc, with `pip install pandoc-shortcaption`
    - Git, for version control.

2. https://github.com/tompollard/phd_thesis_markdown/fork

3. Clone the repository onto your local computer (or download the Zip file).

4. Navigate to the directory that contains the Makefile and type "make pdf" (or "make html") at the command line to update the PDF (or HTML) in the output directory.
   In case of an error (e.g. `make: *** [pdf] Error 43`) run the following commands:

   ```
   sudo tlmgr install truncate
   ```

```
sudo tlmgr install tocloft
sudo tlmgr install wallpaper
sudo tlmgr install morefloats
sudo tlmgr install sectsty
sudo tlmgr install siunitx
sudo tlmgr install threeparttable
sudo tlmgr update l3packages
sudo tlmgr update l3kernel
sudo tlmgr update l3experimental
```

5. Edit the files in the 'source' directory, then goto step 4.

What else do I need to know?

Some useful points, in a random order:

- each chapter must finish with at least one blank line, otherwise the header of the following chapter may not be picked up.
- add two spaces at the end of a line to force a line break.
- the template uses John Macfarlane's Pandoc to generate the output documents. Refer to this page for Markdown formatting guidelines.
- PDFs are generated using the Latex templates in the style directory. Fonts etc can be changed in the tex templates.
- To change the citation style, just overwrite ref_format.csl with the new style. Style files can be obtained from citationstyles.org/
- For fellow web developers, there is a Grunt task file (Gruntfile.js) which can be used to 'watch' the markdown files. By running `$ npm install` and then `$ npm run watch` the PDF and HTML export is done automatically when saving a Markdown file.
- You can automatically reload the HTML page on your browser using LiveReload with the command `$ npm run livereload`. The HTML page will automatically reload when saving a Markdown file after the export is done.

Contributing

Contributions to the template are encouraged! There are lots of things that could improved, like:

- finding a way to add short captions for the tables, so that the lists of tables can be automatically generated.
- cleaning up the Latex templates, which are messy at the moment.
- improving the style of Word and Tex outputs.

Please fork and edit the project, then send a pull request.

Appendix 5: Initial project specification and plan

During the development of the project initial specification and plan was completely reworked. Thus, following information is deprecated or out of date.

Functional requirements

To create new back end for Frankjnr:

- To create Abstract Machine which supports newest Frankjnr implementation, representing the state of Frank execution process; Compiler which works with implemented Abstract Machine and compiles Frank code to working JavaScript which can run in the browser;
- To facilitate client-side communication of events and DOM updates between Frank code and the browser;

Non-functional requirements

- To create tests which should always pass, in order to test new releases of the system;
- To measure performance (in comparison with the existing back-end and with other kinds of generated JS);

- Easier client-side programming (example, complex parser of a text field);

Technologies

Haskell - main functional programming language to be used;

JavaScript - main client-side language to be used;

Frank - built on Haskell, targeted language by the compiler;
https://github.com/cmcl/frankjnr

Stack - Dependency manager and build tool for Frank;
https://github.com/commercialhaskell/stack

Vole - Compiles functional language to JavaScript;
https://github.com/pigworker/Vole

Shonky - Current Frankjnr compiler;
https://github.com/pigworker/shonky

GHC - Haskell compiler;
https://github.com/ghc/ghc

Software development process

- Gathering requirements and working on project specification (week 1);
- Learning, researching Frank and any other related technologies (week 2);
- Understanding the implementation to reach the end goal (week 2-4);
- Updating project specification, plan if necessary (week 4);
- Working on project architecture, to be able to meet all the requirements (week 5);
- Working on the implementation (week 5 - 11);
- Testing (week 11);

- Project evaluation (week 12);
- Final report and presentation (week 12);
- Maintenance, if necessary (week 12+);

During every stage of the software development process I will update my progress log, which then I'll include in the final report. Also, I'll try to implement some tests during the software implementation stage. And, finally, all code will be stored in a github repository, where it will be well documented and publicly available.

Project evaluation

Project will be evaluated by other researchers who have a clear view of how the software should function. They will test the system and give their feedback and assessment. Evaluators will expect that the existing body of Frank examples should work in my implementation of the system the same way they do in other implementations, as well as client-side programming should become easier, example, writing parsers for data in web forms. Furthermore, project will use continuous evaluation technique, so it will be evaluated, for example, every two weeks by at least one researcher, in order to follow Agile software development practices.

Risks

- Dependence on Frank implementation (Frankjnr);
- Estimating and scheduling development time. As I have never worked on any similar project before, correctly estimating and scheduling time might be difficult;
- Planning before fully understanding the implementation process. For example, there might be conflict requirements or just in general incomplete specification as I don't yet have full understanding of technologies I'm going to use.

References

B. C. Pierce and D. N. Turner, 2000. *Local type inference*,

Daniel Hillerström and Sam Lindley, 2016. *Proceedings of the 1st International Workshop on Type-Driven Development*,

Dexter Kozen and Carron Shankland, 2004. *Mathematics of Program Construction*,

F.V. McBride, 1970. *Computer Aided Manipulation of Symbols*,

G. D. Plotkin and J. Power, 2001a. *Adequacy for algebraic effects*,

G. D. Plotkin and J. Power, 2003. *Algebraic operations and generic effects*,

G. D. Plotkin and J. Power, 2009. *Handlers of algebraic effects*,

G. D. Plotkin and J. Power, 2002. *Notions of computation determine monads*,

G. D. Plotkin and J. Power, 2001b. *Semantics for algebraic operations*,

G. D. Plotkin and M. Pretnar, 2013. *Handling algebraic effects*,

Graham Hutton and Joel J. Wright, 2004. *Compiling Exceptions Correctly*,

Lennart Augustsson, 1985. *Functional Programming and Computer Architecture*,

Sam Lindley, Conor McBride & Craig McLaughlin, 2016. *Do be do be do*,

Thomas Jonson, 1985. *Springer LNCS 201*,