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

Run Frank in 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:
Professor Conor McBride

University of Strathclyde, UK
February 2017

Abstract

Frank is a strongly typed, strict functional programming language invented by Sam Lindley and 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

Abbreviations

API	**A**pplication **P**rogramming **I**nterface
JSON	**J**ava**S**cript **O**bject **N**otation

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 Prof. Conor McBride and Prof. 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.

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; the development tools and languages chosen. 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, describing how it was evaluated and to propose relevant future work.

Chapter 2

Related Work

2.1 Shonky

Shonky is untyped and impure functional programming language. The key feature of Shonky is that it supports local handling of computational effect, using the regular application syntax. This means one process can coroutine many other subprocesses. In that sense it is very similar to Frank, just without type support. Its interpreter is written in Haskell, although it has potential to be ported to JavaScript or PHP to support web operations.

In the context of the project Shonky language data structures are used by the Abstract Machine, in Chapter 5 it is explained in great detail of how exactly were they used and for what purpose. 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 an 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 the user is able to use Frank to write expressions. After parsing Frank code

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

2.2.1 Frankjnr limitations

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

2.3 Vole

Vole is lightweight functional programming language with its own Compiler and two Abstract Machines. One Machine is written in Haskell and provides opportunity to run compiled Vole programs on the local machine. Other Virtual Machine is written in JavaScript, because of this it is possible to run Vole programs on the web, there are 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 application front-end on run-time.

In the context of this project - Vole is a useful resource, because it essentially tries to solve the same problem but for a different language. Author had to study Vole, to understand its technical implementation, usage of resources and to expand his knowledge of Compilers and Abstract Machines. Lastly, Vole is used for evaluation purposes as another benchmark comparison.

2.4 Conclusion

Three main related projects are Vole, Shonky and Frankjnr. Vole was used as an working example of a solution for similar problem, which got author thinking of different ways to approach the problem. Shonky and Frankjnr were used directly, because Frankjnr is the newest implementation of Frank

language and it uses Shonky's interpreter for executing Frank programs. Therefore, the idea was to take existing Frankjnr and replace the Shonky's interpreter with 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 problem, challenges, requirements and involved risks.

3.1 Problem overview

The main aim of the project was to develop new Compiler and corresponding Abstract Machine for existing functional language Frank implementation called Frankjnr. The difference from the old ones was that new Compiler and Abstract Machine would support web development. Many different challenges follow from that. First one being authors initial lack of knowledge on Frank language. Frank being full and complex language author had to study it and overcome any difficulties regarding the language by researching the paper on Frank "*Do be do be do*" (Sam Lindley, Conor McBride & Craig McLaughlin 2016).

A second challenge was overcoming the complexity of existing systems. Because this project is not standalone, it strictly depends on the Shonky Syntax implementation as well as Frankjnr handling Frank's code. Author had to take time and carefully research existing systems, to understand how ev-

erything fits together and how efficiently replace exiting Shonky interpreter with new Compiler and Abstract Machine.

Another challenge was the difficult nature of compilers and abstract machines. This was the most difficult challenge to overcome due to the depth and the amount of detail each of them have. Thus, the incremental approach was adapted, in order to for the author to learn step by step, starting with simpler things. Author started the project by developing experimental system with its own language,compiler and virtual machine to understand the crucial concepts of compiler and abstract machines.

Fourth challenge was testing and validation. Because the project is focused on developing completely new system without any usage of frameworks, testing framework had to be developed which could run test cases and check their correctness without any interaction of the user. Testing framework was developed by utilizing Bash Script, Expect Script languages and Node, Webpack, Ghci commands. It was firstly, created for experimental system and then based on observation and testing, altered and improved for the final system. In the end, it dramatically improved development time by constantly locating bugs and validating the correctness of the outputs.

The project was extremely broad and involved great deal of initial research as well as constant analysis of next steps, thus development was slow but effective. However, due to time constrains development time had to be planned very carefully, because of this some acknowledged directions of development had to be ignored to finish the prototype in time. For example, one of those directions was Haskell enforced type checker. Haskell compiler could have had a type checker which would prevent any bugs regarding generated JavaScript programs by enforcing its types. Now such bugs are spotted by the console window of the browser or testing framework.

3.1.1 Project risks

- Dependence on Frank implementation (Frankjnr). Instances of Frank code will be tested to make sure it behaves as expected;
- Estimating and scheduling development time. Due to authors inexpe-

rience with the subject of the project, correctly estimating and scheduling time is difficult. However, meetings with supervisor are organized regularly, to make sure the project is on track;
- Lack of available resources;
- Authors lack of experience with languages being used and analyzed concepts (compiler and virtual machines).

3.2 Requirements Analysis

Before any development could begin requirements needed to be gathered, author did this by discussing how the system should behave and what technologies it should utilize with the supervisor. He, also, attended programming languages seminar where Frank language was discussed, gaining different perspective on the whole project. Lastly, author read papers on relevant topics, such as "Compiling Exceptions Correctly" (Graham Hutton and Joel Wrigh 2004), to expand his knowledge and gain more insight in what to do. Further research was made to ensure the appropriateness of chosen languages and technologies.

3.2.1 Development Tools and Languages

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

3.2.2 Vagrant

Vagrant is an optional tool to create separate development environment for the project. It runs on Virtual Box and it is essentially a server on your local computer which you can boot up and work on. Furthermore, Vagrant comes with up to date relevant libraries out of the box. Author used it to avoid installing software and managing libraries on local machine, thus speeding up development.

3.2.3 Webpack

Webpack is a module builder and it is available as npm package. In this project it is used for Abstract Machine development as it adds some wanted features to plain JavaScript, and in the end everything is compiled to a single light JavaScript file. The most important is an ability to create and export different modules, for example, module could be a function or a variable. This lets for improved project structure as you are able to keep JavaScript components in separate files, thus making it easer to develop and navigate through code.

3.2.4 JavaScript

JavaScript is a client side scripting language. In this project output of the compiler is generated in JavaScript, which is then used by Virtual Machine which is, also, written in JavaScript so that both could 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. And finally, there are lots of libraries and features to support web development.

3.2.4.1 Alternative - CoffeScript

CoffeScript is much newer language and it improves upon JavaScript syntax, allowing for neater declarations, introduces new features and reinforces the structure of the code. CoffeScript can run in any environment where JavaScript can run, because in the end it is compiled to JavaScript. It is available as npm package. However, it wasn't used because author didn't want another layer of compilation going on in the background.

3.2.5 Haskell

Haskell is a statically, implicitly typed, lazy, standardized functional programming language with non-strict semantics. Haskell 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, like pattern-matching, efficient recursion, support for monadic structures. Moreover, "Frankjnr" and "Shonky" are written in Haskell as well, thus, using Haskell would provide easier compatibility with those projects.

3.2.6 Report Markdown

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

3.3 Specification

3.3.1 Functional requirements

- To create new back end for Frankjnr implementation:
  - Develop Compiler which uses Shonky's language (Syntax file) and outputs a sensible and correct JavaScript code structure;
  - Develop 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 set of tests which should always pass, before each new release of the system. Test cases should be increased after every iteration to validate new features and ensure the previous features are not broken;
- To develop testing framework, which tests the system using the list of predetermine 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 (example, complex parser of a text field).

3.3.3 Use Cases

Use cases are not really relevant to this project, since the developed compiler and abstract machine support full Frank language, which potentially can be used in infinite number of ways. But generally, the user puts sensible Frank code into a Frank file then initiate compilation by using project's compiler, the compiler will generate a JavaScript file which user can include in their web project. Furthermore, user needs to include the machine, which utilizes previously compiled code, into their project. For more detailed information on how to use the system, check usage instruction 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 it were the difficult nature of the project and author's initial lack of knowledge on the subject; this methodology allows the developer to focus on few features at a time building the system incrementally, thus making it easier to learn as you go.

Iterative and incremental development (IID) is a process which grows the system incrementally, feature by feature during self-contained cycles of analysis,

design, implementation and testing which end when the system is finished. Meaning, the system is improved (more functionality added) through every iteration.

The core benefits of this methodology are:

- Regression testing is conducted after each iteration, where only few changes are made through single iteration, because of this faulty elements of the software are easily identified and fixed before starting the next iteration;
- Testing of systems features is a bit easier, because this methodology allows for targeted testing of each element within the system;
- System could easily shift directions of development after each iteration;
- Learning curve is much flatter than usual, because developer is focusing on few features at a given time.

Another big decision was to start with simpler system, an experimental throughout which the author could learn the basics and adapt gained knowledge in the development of the final system. The experiment took about 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 initial author knowledge in the field, the most optimal plan was to start with small, less demanding development and expand gradually. The intricacies consist of:

- Frank is a complex functional language;
- "Frankjnr" adds another layer of complexity, since author needs to be aware of the implementation and 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;
- Complexity of the compilers and their implementation;
- Complexity of abstract machines and their implementation;

Thus, simpler language was developed with matching compiler and abstract machine. Both, the compiler and the machine were developed while keeping in mind that their key parts will be reused for the final system. So efficiency, reliability, structure were all important factors.

4.2 Experimental system

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, which syntax supports few specific operations, such as, sum of two expressions, Throw & Catch, Set, Next, Get, new reference. Each of the operations were carefully selected, where their implementation in the abstract machine varies significantly. Thus, offering broad and important learning experience.

Full language definition

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

```
| Val Int
```

Definition of 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 was raised (something went wrong), then the second expression will be evaluated.

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

Syntax definition of "WithRef" command. It creates new reference with a given value. First string variable of the command defines name of new reference, second value is an expression which defines how to compute initial value of 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.

```
  | WithRef String Expr Expr
```

Syntax definition of getting the value of a defined reference.

```
  | Get String
```

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

```
  | String := Expr
```

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

```
  | Expr :> Expr
```

4.2.2 Compiler

Purpose of the Compiler is to take in an formatted expression of the experimental language described above and output a JavaScript compiling data structure (array of functions), which could be used by the abstract machine.

Below is a definition of code generation monad. It contains a constructor "MkCodeGen" and a deconstructor "codeGen". It takes in an next available

integer value and outputs a data structure which holds a list of integer which map to JavaScript code (string), next available number after the compilation and 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 it is used by compile function to make new function definitions.

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

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

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

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

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

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

4.2.2.1 Formating and outputting to file

Output formating is done by "jsSetup" function. It takes the name of the array, the output of "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 everything to a file - "generated.js", which can be safely 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

Purpose of the abstract machine is to take in a compiled program and provide semantics for the file to be runnable in 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 part 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

Each generated data structure 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 array of resumptions ready to be used by the abstract machine. The semantics of this particular structure are simple, to add two numbers "2 + 3", so the expected output is "5". Comments in the snippet below explain meaning of different variables in the structure. For more complicated examples see *main/example_programs*.

```
var ProgramFoo = [];

ProgramFoo[0] = function (s) {
    return {
        stack: s, // stack
        tag: "num", //expression type
        data: 3 // expression value
    }
};
ProgramFoo[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 detailed implementation of the experimental abstract machine. It is defined as a function which takes in array of

resumptions as an argument.

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

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

Each resumption is function, which takes in a stack as a parameter and return a mode. Abstract machine will finish compilation if "mode.tag" is not equal to "go". However, if it is equal to "go" then mode must be reinitialized, because not all instructions are done, by getting creating the next mode from resumptions array and passing stack to it, so that the mode has access to previous stack.

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

After the mode is reinitialized "mode.tag" can not be equal to "go" and mode's stack can not be empty. If these requirements are not met, the next mode will be retrieve, however, if there are none left, the abstract machine will stop executing and return the last mode.

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

If the execution is still going then the behavior of the Abstract Machine will differ based on mode's "tag" parameter. It could be equal to these values:

- **“num”** - all of the basic evaluations of given expression:
  - Addition (“left” and “right” tags) - “left” tag creates a stack frame with tag “right”, thus preparing a frame for addition. If the top of the stack “tag” is equal to “right” it means that abstract machine can add two of the top frames together, because “left” was previously called and the values of frames are prepared to be added.
  - “Catch” - creates a stack frame with 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 tag “WithRefRight”, it is different from other stack frames because it has a “name” parameter, which is needed to identify 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 “:>right” the abstract machine will take its data without adding anything and it will delete the previous stack frame. These operations evaluate two expressions but return the value of the second.
  - Set (“:=”) - very similar to “Get” command (described below), key difference is that it alters the value of stack frame which has tag “WithRefRight” with given name of the reference. This command utilizes linked list stack saving structure to be able to restore the stack while saving any changes made. 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; 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 it means exception was handled, therefore the abstract machine will reinitialize mode to continue executions by taking "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, output's either a value of the reference if it does find it, or tries to throw an exception if it does not. It, also, utilizes 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. And finally, Abstract Machine outputs final value of the execution on the separate line for testing purposes, it is used by testing framework to check for expected and actual output of a test.

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

4.2.3.2 Linked Stack Saving and Restoring

Abstract machine uses "saver" helper function in "get" and ":=" implementations. This function lets the machine to inspect the depths of the stack and to assign new values to existing frames of the stack by remembering changes made. To achieve this, abstract machine saves each frame of the stack by linking frames together, thus each newly saved frame has a link to previous frame. Below "saveStack" function is displayed, the "m" variable represents the current stack frame and "prev" field is a link to previous saved frame.

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

The save is 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 stack is successfully restored, all of the save data must be destroyed and parameters reseted to keep future saves unaffected. Finally, current limitation is that only one instance of stack save can exist at a given time.

4.2.3.3 Final Remarks

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

Room for improvement:

- Efficiency and optimization, for example, by removing "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 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 compiled expression array and managing its resources, such as mode, stack and saved stack.

Key things to improve are optimization & performance.

The further plan was to roll out the lessons learned creating compiler and 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

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

5.1 Introduction

Final system started to develop on week six of the second semester after the end of experimental system development. Some parts of the code were lifted from the earlier experiment and main concepts of how virtual machines and compilers should work are reused.

5.1.1 Disclaimer

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. So the whole Frankjnr project was used and new back end was placing 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, updates 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;

More detailed descriptions can be found at 'Related work' section of the report or at their respective Github pages.

5.2 Project folder structure

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: Project 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
	webpack.js	Webpack configuration file

Paths and Files		Summary
	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

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

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

Developed compiler constatly uses Shonky data sctructures, located in *Syntax.hs* file, because it receives a list of program definitions from Frankjnr compiler in a form of Shonky syntax data structure, in particular - "[Def Exp]". "Def Exp" can be either ":=" or "DF", however Frank compiler always gives back list of "DF", thus this compiler will only support them as well. "DF" means functions definitions and they consist of:

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

Goal for the compiler is to compile data type "Exp" (type of expression, such as atom or application) to working JavaScript, generating functions that do intermediate steps. The code for compiling "Exp" will expect a lookup table which maps the "environment" array (stores values of pattern values) to array indices. In order to find out if certain values are in scope, otherwise compilation will fail.

```
type EnvTable = [(String, Int)]
```

Furthermore, there are two types of patterns. Pattern for computation - "Pat" and pattern for value - "VPat". Patterns for computation can be thunk, command or "VPat", such as variable "VPV", pair "VPat :&: VPat" and so on. Compiler has to build an "EnvTable" from these patterns, hence these two functions: "patCompile" and "vpatCompile". They have been implemented using counter monad "Counter", which is there to count each patern

and is used for “environment” lookup table. And the core pattern maching priciple used was “match-this-or-bust” described in PhD thesis “Computer Aided Manipulation of Symbols” (F.V. McBride 1970). Therefore, these functions will generate series of checks for each individual pattern and if everything is fine and the pattern is a variable and not, for example, an atom or integer, add them to the “environment”, else it will throw an expection. In the code snippet below the patern is an integer so we don’t need to add them to the environemnt, but it still has to pass these tests in order to match. Value “next” is just a counter to arange correct indeces for “environment” lookup table:

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

After “environment” lookup table is compiled and ready to go, compiler now needs to use it to compile expressions - “Exp”. This is achieved in “expCompile” function, which takes in an “environment” lookup table, function lookup table, stack (string data structure), expression 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 indeces 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 instance “EA” (atoms) type is straightforward, because atoms are simple and compiler only needs to compile the actual atom. It looks like this:

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

However, for example “pair” type of expressions are more complicated, because the “pair” contains two expressions, so the compiler has to compile them both separately:

```
expCompile xis ftable stk (ecar :& ecdr) = do
  fcdr <- expFun xis ftable ecdr
  expCompile xis ftable
    ("{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 do this it utilizes the helper function named “tailCompile”, which recursivly builds a linked list.

However, before any of individual expression compilation or “environement” building can begin, compiler needs to combine everything into one JavaScript data structure, forming resumptions and operators arrays in the process. The function for this is “operatorCompile”, which initiates chain reaction of function calls for each function definision and concatines the results into one data structure. This chain reaction of function calls consist of “oneCompile”, which starts to compile single fuction definition and forms an “operators” array entry. Then it calls “makeOperator”, which sets the “interface” (all available commands for given operator) by calling “availableCommands” and “implementation” (JavaScript function definision) by calling “funCompile”. “funCompile” is responsible for forming the actual function definition of the operator and it procedes to initiate “linesCompile”, which forms a “try” and “catch” blocks and calls the final function “lineCompile”, who fills the lines with compiled expression data by actually calling “patCompile” and “expCompile” functions with data on functions patterns and expressions; finally, “lineCompile” forms a return statement of the function. To see how these expressions and paterns look compiled, see “gen.js” file.

5.3.1 Helper functions

This section will briefly explain functionality of few helper functions.

parseShonky - is used for testing purposes, it takes Shonky syntax file (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 - wraps everything into one function, it takes the output of "operatorCompile", formats it and writes the result of the compilation to the "gen.js" file.

5.4 Abstract machine

Purpose of the abstract machine is to take in generated output of the compiler and run it in the web, thus completing the task of running Frank code in the browser. For full usage & 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.

5.4.1 Implementation

Abstract machine takes contents of gen.js as an input, which contains two different arrays: operators and resumptions. Operators are equivalent to functions in Haskell, resumptions are computations waiting to be executed. In current implementation starting operator is always the first function, so main method should be at the top of file. And they return modes which are computation who store stack, here 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 machine check the current computation tag, it can either be a "value" or a "command" and depending on which one it is, machine will act accordingly.

5.4.1.1 Computation - "value"

If the "mode.comp.tag" is equal to "value" then 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 new mode building or reducing the stack in the process (look at *JavaScript type definitions* section to see their structure):

- **"car"** - will construct mode out of calling a resumption based on "mode.stack.frame.cdr" value. For resumption to construct a new mode they need 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.

  ```
  frame: {
    tag: "cdr",
    car: mode.comp.value
  },
  ```

- **"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 function 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 argument list is not empty machine needs to construct new mode with frame tag “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 to keep the function that will be applied when the arguments “waiting” list is empty. “env” to keep the environment, “ready” list to know which arguments have been parsed, initially it is empty. “waiting” to keep track of list of arguments that are left unchecked. And, lastly, “handles” with “waitingHandles” 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 list. And the initial handle list is extracted from operators interface with 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).

5.4.1.2 Computation - “command”

Because of “command” type of computations Frank is different from other functional languages. When command computation appears the current ex-

ecution 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 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. Each time it doesn't find it, it will place top frame in the callback and move down by one frame:

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

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.

5.4.2 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 that 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 call apply function, if it is not then create new "arg" frame in the same fashion as in "fun" option. The key difference is that "fun" could be instantly applied if it didn't have any arguments, thus not creating any "arg" frames. Moreover, there is 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, and the machine is turning it into top level operator, concept introduced by (Thomas Jonson 1985) and it is named "Lambda Lifting". Therefore, mode is constructed out of an operator variable.
- **"operator"** - Means top level function, mode is constructed from an operator depending on "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. Thus new mode looks like this:

```
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 looping through the "callback" while building the stack with callback's frames. When it is done, machine returns a mode constructed of the restored stack and first argument.

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

printer - takes the mode of finished execution, parses it to make it readable and displays it on the screen.

5.4.3 JavaScript type definitions

This section explains the JavaScript types used to enforce the data structure of programs.

"JSRun" is an operator, who always return mode. They are located in operators array in the generated program ("gen.js"). Mode has a stack and current computation; top stack frame and current computation both determine what to do next.

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

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. This structure is used regularly throughout the project.

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

Stacks frame type, which determines the operation that needs to be done when current computation is equal to "value".

```
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 }
```

List data structure, where it could be empty or have an current element and tailing list of elements.

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

Computation could either be a "value" or a "command".

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

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

These are the types of what value can be. “atom” is just an value that cannot be deconstructed any further. “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 definition of command that it was looking for. “thunk” represents a suspended computation. And “local” means local function, which do to procedure called “Lambda Lifting” (Thomas Jonson 1985), abstract machine will turn it into a top level function 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}
```

5.5 Built in functions

Current built in functions are “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 compared to others which are generated. Compiler is not able to give them meaning because he 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. 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 list of integer variables "y", which initially does not make any sense. However, machine has encoded semantics for this situation, therefore if integer is applied to a list of arguments that could only mean one of two things, either that integer needs to be added to first element of the list or substracted from it. Machine determinse this by looking how many elements are in the list and applies the correct opperation 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. This would skip checking all frames, which tags are not equal to “arg”, therefore improving performance.

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.

Chapter 6

Verification and Validation

This project's verification and validation were done strictly throughout testing. Development was done in two parts, development of experimental system (for learning purposes) and development of final system. Initial testing started at very early stage of the project, by testing experimental system's abstract machine behavior without the interaction of the compiler. At that point it was done manually, by the author typing in expressions and looking through the output, however as development went on it quickly became too inefficient. Thus a experimental testing framework was developed (February 10th, 2017), which would go through pre-set test cases one by one without any human interaction; giving feedback for the user in the process. Author then adapted the system and applied lessons learned to create similar but improved framework for the final system.

During the implementation of both systems test cases were added regularly, after each newly developed component. Moreover, on account of iterative and incremental development methodology (IDD) author was able to write targeted test cases for every component of the system. Because of this, new bugs were identified and tracked faster, thus leading for faster software development. Test cases were initiated before every content push to git and during development to check for any broken parts of the system.

After the implementation has ended, test framework was initiated one last time to see if all tests still pass, as well as some manual testing was done

by the author 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 are the same in both systems. More on this in **Chapter 7**.

Possible improvement would to be to use service similar to *Jenkins*, in order to automate tests even more by automatically launching them and giving back feedback. But only on certain conditions, for example, after every push to git or once a day regularly.

For 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: Bash script and Expect script. Its purpose is to automate the testing process. Below each of these scripts will be reviewed.

6.1.1 Bash script

Bash script is the main script in the testing framework which stores all test cases, then goes through them one by one. To launch it simply type *./tester.sh* in the terminal window. For full guide on installation and usage see **Appendix 2**.

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

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

For each test case Bash script launches Expect script and passes test case parameters to it, in particular it passes 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 new program is generated, system must recompile abstract machine's code to use newly generated program. *Webpack* is used to compile JavaScript files to one and to manage their structure.

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

After successful recompilation of JavaScript Bash script has to retrieve the output of the program by getting the last line of the console output, it does this by utilizing Node functionality and some string manipulation.

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

Finally, Bash script just compares the expected output with actual output and gives back the result for the user to see.

```
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, in this case *GHCI*. 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 which was passed by the Bash script.

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

After that it launches *GHCI* terminal.

```
spawn ghci
```

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

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

Finally, Expect script sends the two following commands with some variables (taken form 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 test framework computation;

6.2 Final testing framework

Contained in *final_implementation/Backend/tests* folder. Framework is designed to launch number of programs, located in *test_cases* folder, and give back feedback to the user, which test programs succeeded and which ones failed. This framework eliminates manual testing, lets identify bugs faster, thus speeding up development process and easing maintenance. And it is fully written in Bash Script.

Some parts of the test framework were lifted from earlier experiment, however the key differences are that it is not using GHC compiler directly; so it eliminates the need for Expect script (previously located in "helper.sh" file),thus improving performance of the framework. Another change is that test cases don't take in expressions anymore, instead they take in paths to actual programs, therefore providing ability to launch them and check their outputs.

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

6.2.1 Implementation

Similarly like in experimental system's testing framework, test cases are stored in array. Each of them are objects which store three values: path of the test program, name of the test and 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 recompiled to include new "gen.js" file.

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

Then it just retrieves the output with *Node* and checks if it as expected or not; letting the user know if test failed or passed (same way as in experimental testing framework). Finally, some statistics are shown on how many tests in total have passed and failed with corresponding percentages.

6.2.2 Possible improvements

- Providing more statistics when all tests are executed;
- Timestamp tests, letting the user know how much time it took to run individual tests and all of them together;
- Improve performance by not trying to launch programs which fail to compile by "Frankjnr" compiler.

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. Experimental system is able to compile and run in the browser specific list of operations, in particular sum of two expressions, "Throw" and "Catch", "Set", "Next", "Get" and to create new reference. These operations were chosen because they each have significantly different implementation, thus offering broader learning experience for the author. Moreover, abstract machine is written in JavaScript, therefore it is able to communicate with compiled expressions on run-time. Defining quality of the abstract machine is that it utilizes stack as its core data structure, with support for interrupts. They have connected handlers who when initiated start going through the stack, in order to execute some operation; when they are done, 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 of sum of previous "x" value and value "11". Moving on

"Next" (":>") operation is used to initiate last expression, where the new value "x" is taken and added to value "30". 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, computation is interrupted on ":=" and "Get" commands to search through the stack and retrieve or update reference of "x".

7.1.2 Final system

Lessons learned, along with the code style and a few functions of experimental system, were adapted in creation of the final system. Having done this, 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. Final system supports variables, integer values, atoms, pairs, function application, local functions, commands, thunk patterns, top level functions and contains built in semantics for addition and subtraction. All these features combined offer close to full development package, thus almost any current Frank program could be compiled by the new compiler and successfully executed by the abstract machine in the browser.

Quick example of how the compiler and abstract machine work together and how user may use it. User is able to execute Frank programs in four steps. First step is to write a program in Frank with file ending of ".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)
```

Second step is to compile it with flag "output-js" to initiate new compiler. This will generate "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
```

Third step is to recompile the machine's code (in *machine* folder), to include newly generated file, with *webpack* command. And finally user needs to include "output.js" form *dist* folder, in his project's HTML file. In this example 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 present because of time constrains.

- Compilation of Frank programs with 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 Frank program must have a "main" function and it must be the first one in the file;
- Lack of web functionality. Even though Frank language can be used to code in web context, current implementation does not have any web features implemented because of time constrains;
- Some features are still in prototype stages and very small amount of them are still not implemented, such as string concatenation.

Moreover, the system can still improve on a variety of concerns, such as performance and building procedures as they were not initial goals of the project.

7.2 Evaluation

Experimental evaluation was chosen for this project and it was done in two parts. First part was benchmarking performance of developed system against two other similar systems (automatic evaluation). Second part was comparison of functionality between developed system and Frankjnr original components.

7.2.1 Benchmark

Benchmarking framework can be found in *final_implementation/Backend/benchmark* folder. Tests can be initiated in a terminal window by typing *./evalNewBackend.sh*, *./evalVole.sh* or *./evalFrankjnr.sh*. Requirements for launching the benchmark tests are the same as for the final system, in addition the scripts must have permissions.

This benchmark was solely focused on the performance of the systems. Two type of times were taken into account: compilation time and execution time. Compilation time is how fast given program is parsed and compiled; execution time is how fast virtual machine returns the final result. Vagrant (Virtual Box) environment was used for running the tests with two gigabytes of allocated memory. These environments are known to be considerably slower than real ones, therefore chosen systems might perform faster in real world, however in this case systems were only compared between each other. The following systems were benchmarked:

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

Single test included compilation and execution of a given program hundred times and each system were taken 5 times. The program, which was executed had same semantics for all systems. Below the program is shown which was used to benchmark Fraknjr original and new system.

```
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.

Concerns and challenges were:

- Vole uses different compiler which has its own terminal, thus expect script was used. This might have affected performance.
- Final system is dependent on Frankjnr compiler.
- Final system uses JavaScript powered machine compared to other which use Haskell for abstract machine.

This evaluation gives a clear idea of the state of the system in terms of performance compared to other similar systems. However, these results were expected, since the performance was not one of the core goals of the project. Moreover, due to developed benchmark framework, it could be easily used again after further development to quickly check for changes in performance.

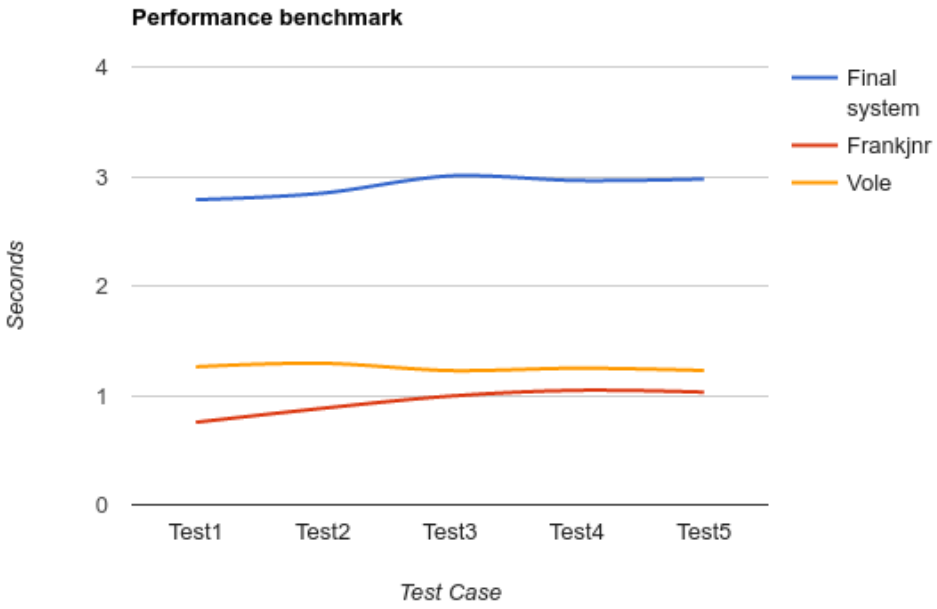

Figure 7.1: Benchmark result graph

7.2.2 Functionality comparison

One of the main goals of the final system was to support the same functionality as Frankjnr original components do and implement addition web development features on top. Known main features of both systems were extracted and compared. The table below shows the results 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

Feature	Final system	Original system
Application	Yes	Yes
Composition	No	Yes
Built-in functions	Yes	Yes
Parse Shonky files	Yes	No
Web development support	Yes	No

New test program was created and executed by both systems to collect the results for each feature. If the program would throw an exception or the output was not as expected, the tested feature is not supported by the system. Furthermore, author has taken Frankjnr original test cases from *tests* folder and initiated them against the final system, in order to see the results from different perspective. Out of twelve original tests, four have not passed on the final system. However, all of them were connected to functionality, witch was not implemented by the final system at the time, for example, string concatenation. For more detailed overview of testing, see **chapter 6**.

This evaluation evidently shows that developed system can directly compete with the original system and that most goals of the project were reached.

Chapter 8

Summary & Conclusion

8.1 Summary

Overall, the outcome of the project was successful. Author developed two systems with their own matching compilers and virtual machines. Initial system was developed as an experiment, to provide the author with valuable knowledge on the subject, witch then he applied into development of the final system. The system is able to compile Frank code and run it in the browser, furthermore, almost all of current Frank programs can be compiled and executed successfully by the system. Therefore, the main goal of the project was obtained. However, due to time constrains, final system is very much in prototype stages as it still does not support all of Frank's features and contains few fixable limitations, highlighted in **Future work** section.

By completing the project author was able to learn variety of concepts. Firstly, 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 infrastructure, functionality and implementation of compilers and abstract machines by researching and participating in numerous discussions with the supervisor, Conor McBride. Author, also, gained greater insight on Bash script while creating testing frameworks for the systems and working on project evaluation. And, finally, author learned the language of Frank and implementation of its defining

qualities.

8.2 Future work

- Implement missing features, such as string concatenation;
- Improve testing framework to show more statistics and use timestamps;
- Expand support for web features, such as HTTP request handling support;
- To improve abstract machine's performance by storing stack frames in chunks. This would speed up searching and restoring;
- Enforce JavaScript type definitions with Haskell structures, to decrease the risk of bugs;
- Increase compiler efficiency by updating pattern matching procedures to building a tree of switches described in "Functional Programming and Computer Architecture" (Lennart Augustsson 1985);
- Improve building procedures;
- Further optimize compiler and 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:

March 16:

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 sections
March 20:
Report:
Testing and validation sections
March 21:
Benchmarking and evaluation sections
March 22:
Evaluation and conclusion
Appendix 2: Usage & installation instructions {.unnumbered}

Experimental test framework

Pre-requirements

- Node;
- npm;

- Webpack;
- GHC;
- Expect Script - install by typing this command:

```
apt-get install expect
```

Usage

To run simply type in the terminal window:

```
./tester.sh
```

Troubleshooting

If files don't have permission, do:

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

Final test framework

Pre-requirements

- Node;
- npm;
- Webpack;
- GHC;
- Frank - instructions how to setup are on Frankjnr Github *readme.md* fie or in the Appendix 4 of this report.

Usage

To run simply type in the terminal window:

```
./tester.sh
```

You should be able to see tests going through with some feedback.

Troubleshooting

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

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

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: pr 0 1.
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: 1 2 3.

```
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 [0, 1, 2] [0, 1, 2].
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: zero, suc zero, suc suc zero.

```
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 [zero, suc zero, suc suc zero] [zero, suc zero, suc suc zero].
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: [[2], [3], [4]].

```
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

Run Frank in Browser

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.

References

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

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 Wrigh, 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*,