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 strongly typed, strict functional programming language invented by Sam Lindley and Conor McBride and it is influenced by Paul Blain Levy's call-by-push-value calculus. Featuring a bidirectional effect type system, effect polymorphism, and 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 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. So 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 main top-level goal of the project is to compile Frank to JavaScript and run it in the browser. Thus, users would be able to use Frank for web development purposes. This involves creating a Compiler and Virtual Machine (abstract machine) which can support compiled Frank structure.

Acknowledgements

I would like to express my special thanks of gratitude to my supervisor, Conor Mcbride for his guidance throughout this project.

Also, I would like to thank my friends for helping me think with 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, objectives and outcome. Also, in the last section report structure is displayed.

1.1 Background

Functional languages are evolving, they are constantly changing to innovate and adapt to ever changing needs of software engineering. Thus, concept of Frank language was created by Prof. Conor McBride and Prof. Sam Lindley followed by its implementation called Frank and more recently newer release - Frankjnr.

Frank is 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. 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 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 district features of Frank, in particular its capability to support effects and handlers, it has potential to be used in web development context. However, neither of Frank implementations currently support this. Thus this projects motivation is to be able to use Frank for web development. Using functional language for web development could greatly ease the process of writing parsers and form validations in the web. Furthermore, Frank code would be essentially converted into JavaScript because of its support for functional programming. This means Frank would potentially gain all the JavaScript functionality and could even use JavaScript libraries. For example, Frank would be able to handle different http requests, such as GET or POST, Frank could fire events (ex. alert boxes) or handle events (ex. mouse clicks).

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

This projects requires knowledge how Compilers and Abstract Machines work and how to use them together them. Author chose this project while knowing that it would be extremely challenging but most effective learning experience.

1.2 Objectives

- Utilize Frankjnr implementation of Frank and Shonky language.
- Develop Code Compiler which compiles Shonky's code to JavaScript program.
- Develop Abstract Machine implementation which supports the output

of the Compiler.

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

1.3 Project Outcome

1.4 Summary of chapters

Chapter 2 - Related Work

The point of the chapter is to highlight work done by others that somehow ties in with this project. It either may be work that author is basing his work off of, or work that shows others attempts to solve the similar problems.

Chapter 3 - Problem Description and Specification

This chapter briefly overviews the main problem of the project. It expands upon functional and non-functional requirements, followed by detailed specification and explanation of design methodology.

Chapter 4 - Initial development & experimental system

This chapter is focused on initial experimental system. It highlights the purpose of each component, their implementation, drawbacks and any possible improvements.

Chapter 5 - Detailed Design and Implementation of the final system

Chapter 6 - Verification and Validation

Chapter 7 - Results and Evaluation

Chapter 2

Related Work

2.1 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 extremely useful resource, because Vole 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.

2.2 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.

2.3 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.

2.3.1 Frankjnr limitations

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

2.4 Ocaml

2.5 Haste

Haste is an implementation of the Haskell functional programming language, designed for web applications and it is being used in the industry. It supports the full Haskell language, including GHC extensions because it is based on GHC compiler. Haste support modern web technologies such as WebSockets, LocalStorage, Canvas, etc. . Haste, also, has support for effects and handlers. Furthermore, Haste programs can be compiled to a single JavaScript file. Haste wasn't used as a direct resource, however it provided Author with detailed description of another solution to similar problem.

2.6 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 Abstract Machine (interpreter). So 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. Other related work were used as examples of different solutions to overcome similar obstacles.

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 everything 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 resources, supervisor Conor McBride is the only resource of relevant and accurate information;
- Authors lack of experience with languages being used and overall 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. 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 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 a clear choice 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, so it was easy to cooperate 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. Everything is compiled to a single PDF by utilizing 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 mode detailed information on how to use the system, check usage instruction in the Appendix ???.

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 overall project and the lack of initial author knowledge in the field the most optimal plan was to start with something small 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 language;
- 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 used for the final product. So efficiency, reliability, structure were all key factors.

4.2 Simple system

It consists of:

- Compiler - written in Haskell;
- Language - written in Haskell;
- Machine - written in JavaScript;
- Testing Framework - written in Bash and Expect scripts;

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.

Full language definition

This section will focus on the exact definition of the language. Language is defined as Haskell data type. In this case it supports only different types of expressions.

```
data Expr =
```

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 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 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
```

Each of the operations were carefully selected, where their implementation in the Abstract Machine varies significantly. Because the implementation of these operations will be generalized in the final system, for instance, code for sum of two expressions will cope with all arithmetic calculations of two expressions.

4.2.2 Compiler

Purpose of the Compiler is to take an expression of the simple, experimental language described above and output a JavaScript working program, 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 deconstructor codeGen.

```
newtype CodeGen val = MkCodeGen {
            codeGen :: Int -- next available number (for naming helper functio
                        -> ([(  Int          -- this number...
                             ,  String       -- ...defined as this JS code
                            )]
                           ,  Int         -- next available number after compilat
                           ,  val         -- result of compilation process
                           )  -- usually the list of definitions will start wit
        }                     -- the input "next number" and go up to just befo
                              -- the output "next number"
```

MkCodeGen is used to construct a definition in genDef function displayed below. genDef returns the definition number.

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

genDef is used by compile function to make new function definitions. Below is the type of compile function. 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 JavaScript code.

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

compile function is either used to create new function definitions...

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

or to compile expressions to JavaScript.

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

Formating and outputting to file

jsSetup takes a 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 Abstract Machine.

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

4.2.3 Abstract machine

Purpose of the Abstract Machine is to take in a compiled program and run it in the browser. It gradually builds a stack from the given program, where each frame of the stack has a link to another frame. The elegant part of this is that, stack frames can be saved, updated, deleted and restored; thus, making the Machines data structure flexible.

Program Structure

Each program is an array and 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 a compiled program ready to be used by Abstract Machine. This particular program is simple, it only adds two numbers “2 + 3”, so the expected output is “5”.

```
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
    }
};
```

Abstract Machine Implementation

This section will explain detailed implementation of the experimental Abstract Machine. It is defined as a function which takes in a compiled program as an argument.

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 evaluate next function. Finally, the “data” parameter holds an index of next function, so the initial value is the index of last function in a program array.

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

Mode is a function in a program array which takes in a stack as a parameter. Abstract Machine will operate until mode.tag is equal to “go”. If it is then mode must be reinitialized by getting getting a next function from program 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’t be equal to “go” and mode stack can’t be empty. If these requirements aren’t met then it means that execution is over.

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

If the execution is still going then the behaviour of the Abstract Machine will differ based on mode’s tag parameter. Tag could be equal to these values:

- **“num”** - all of the basic evaluations of given expression:
  - Addition (“left” and “right”) - creates a stack frame with tag “right”, 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 all of the other computations are done.

- 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 used stack frame.

- Set (:=) - very similar to “Get” command, key difference is that it alters the stack frame which has tag “WithRefRight” and the given name of the reference. This command utilizes linked list stack saving to be able to restore the stack while saving any changes made. It could throw an exception if the reference is undefined.

- **“throw”** - it defines that something went wrong so the Abstract Machine will look for a “catch” usage in the previous stack, if it doesn’t find it then it will output an exception.

Machine checks if the top of the stack is equal to “catcher”, if it is then it means exception was handled, so Abstract Machine reinitializes mode to continue executions by taking “catcher” values of “stack” and “data”.

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

Else Machine drops the top of the stack and continues to look for “catcher” until stack is empty.

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

- **“get”** - goes through 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 doesn’t. It, also, utilizes linked stack saving and restoring functions, in order to restore the stack if it does find a reference.

After the Abstract Machine finishes running it will output the final stack to the console by invoking a custom printer function for the user to clearly see the stack.

```
printer(mode);
```

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);
```

Linked Stack Saving and Restoring

Abstract Machine uses saver function in “get” and “:=” implementations. This function lets the Machine not only to inspect the depths of the stack but, also, to assign new values to existing frames of the stack. To achieve this, Abstract Machine saves each frame of the stack from top until it finds the frame it is looking for. Below “saveStack” function is displayed, the “m” variable represents the current stack 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. After stack is successfully restored all of the save data must be destroyed and parameters reseted to keep future saves unaffected.

Limitations:

- Only one Stack save can exist at a given time.

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, throwing & catching an exception.

Room for improvement:

- Efficiency and optimization;
- Documentation;
- Functionality;

All of these are addressed in the final implementation.

4.2.4 Testing framework

Testing framework 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.

4.2.4.1 Bash script

Bash script is the main script in the testing framework which stores all test cases, then goes through them one by one.

Sample test case:

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

Complex test case:

```
declare -A test9=(
    [expr]='let xpr = WithRef "x" (Val 22) ("x" := (Get "x" :+: Val 11) :> (Get
    [name]='test_withref_get_set_next'
    [expected]='63'
)
```

Each test case must store three values: expression to be tested, the name of the test and expected output.

For each test case it launches Expect script and passes parameters to it. 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 and generates new program, system must recompile Abstract Machines code to use newly generated program. Webpack is used for all JavaScript code compilations, dependencies and overall structure.

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

After successful recompilation of JavaScript Bash script has to retrieve the output of the program by retrieving the last line of the console output 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
```

4.2.4.2 Expect script

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

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

```
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.

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

Finally, Expect script sends the two following commands 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"
```

Possible improvements for the final testing framework:

- Speed & efficiency;
- Move test cases into separate file;
- More useful statistics at the end of computation;

Usage

To use the testing framework just run ./tester.sh in the terminal window. If there is a permission issue do "chmod +x helper.sh".

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 given program, and managing its resources, such as mode, stack and saved stack.

Key things to improve:

- Optimization

The further plan is to roll out the lessons learned 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

5.1 Introduction

The implementation of final Compiler and Abstract Machine were heavily supervised by Conor McBride.

5.2 Project folder structure

5.3 Possible improvements

use chunks of stacks when top of the chunk is possible needed stack frame and others are not (speedup)

turn local functions into top level operators is called "Lambda-lifting" it was invented by Thomas Jonson in 1985 (Springer LNCS 201). it's used in eg. GHC # Verification and Validation

Chapter 6

Results and Evaluation

Chapter 7

Summary & Conclusion

7.1 Summary

7.2 Future work

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

Appendix 2: Some more extra stuff

References

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

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*,

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