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.

In this project the main goal is to compile Frank to JavaScript and run it in the browser. So, for example, user would be able to edit their MyPlace pages using Frank language. 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.

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. Furthermore, last section, explains the report structure.

1.1 Background

1.2 Objectives

- Develop Code Compiler which compiles Frank 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

2.1 Vole

Vole is lightweight functional programming language with its own Compiler and Abstract Machine. Compiler compiles the Vole code to JavaScript, which can be used by Vole.js (Abstract Machine) and run it on the browser. It has some support for effects and handlers.

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.

2.3 Frankjnr

Frankjnr is an implementation of Frank programming language described in "*Do be do be do*" (Sam Lindley, Conor McBride & Craig McLaughlin 2016).

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.

2.6 Conclusion

Chapter 3

Problem Description and Specification

3.1 Problem overview

3.2 Requirements Analysis

Functional requirements

- To create new back end for Frankjnr:
  - Develop Compiler which uses Shonky's language and outputs a working JavaScript structure of code;
  - Develop Abstract Machine which runs the previously compiled JavaScript code in the browser;
- To facilitate client-side communication of events and DOM updates between Frank code and the browser;

Non-functional requirements

- To create set of tests which should always pass, before each new release of the system;

- To develop testing framework, which tests the system with minimal human interaction;
- To measure performance (in comparison with the existing backend and with other kinds of generated JS);
- Easier client-side programming (example, complex parser of a text field);

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 inexperience with this kind of projects, correctly estimating and scheduling time might be difficult, this, also, greatly increases the possibility of not finishing the project. However, meetings with supervisor are organized regularly, to make sure the project is on track;

3.3 Specification

3.4 Design Methodology

Chapter 4

Initial development & Simple 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 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

```haskell
data Expr =
        | Val Int -- integer value
        | Expr :+: Expr -- sum of two expressions
        | Throw
        | Catch Expr Expr -- evaluated first expression, if result is
                          -- Throw then evaluate second expression
        | Get String -- get the expression of the reference
        | String := Expr -- Set reference value to some expression
        | Expr :> Expr
            -- :> is ugly syntax for ";" (taking value of the second)
        | WithRef String -- name of new reference
                  Expr -- how to compute initial value of new reference
                  Expr -- code that makes use of reference
            --  the WithRef stack frame is the handler for Get and :=
```

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 language described above and output a JavaScript working program which could be used by the Abstract Machine.

```
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"
```

Function compile works together with genDef to make definitions of functions in JavaScript and link them together, by utilizing linked list data structure. Below is displayed a small piece of the compile function and full genDef function.

```
genDef :: String -> CodeGen Int -- make a definition and return its number
genDef code = MkCodeGen $ \ next -> ([(next, code)],next + 1, next)

compile :: Expr -> CodeGen Int -- the Int returned is the entry point
compile e = help "s" e where
    help s (Val n) = genDef $
        "function(s){return{stack:"++ s ++ ", tag:\"num\", data:"++ show n ++"}
```

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.

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 (3 + 2), 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
    }
};
```

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;
- Documentation;
- Functionality;

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

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.

Chapter 6

Verification and Validation

Chapter 7

Results and Evaluation

Chapter 8

Summary & Conclusion

8.1 Summary

8.2 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.

Appendix 2: Some more extra stuff

References

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