CS408 Project Report

Rokas Labeikis

February 4, 2017

0.1 Introduction

Frank is strongly typed, strict functional programming language invented by Sam Lindley and Conor McBride and it is influenced by Paul Blain Levys 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.

So, 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 Virtual Machine (abstract machine) which can support Frank structure.

0.2 The beginning

By choosing this project, I knew that I was stepping out of my comfort zone. At the beginning of the project I was aware only of the purpose of the Abstract Machine, didnt know much about compilers either, or how can they both work together and how to implement either of them. My understanding of Haskell was pretty basic as well. So I knew I have a big challenge ahead of me. Thus, It made sense to begin with something simpler than implementing the whole system from the start. And so, Ive started by developing a simple Abstract Machine with few very basic procedures, just to get me going. Then I had to create a simple language with very few functions, such as addition of two values, Catch and Throw, variable assignment. Furthermore, Ive had to implemented a compiler for that Machine and language, so that they all could work together.

How do they work together? Compiler takes in an expression in my language, such as, (Val 5) and outputs a populated javascript array, which we can call a program. Then the Abstract Machine takes that program and

parses it, populates the stack and starts doing work.

Also, this wasnt for the educational purposes only, I worked on the compiler and Machine by keeping in mind that I would use parts of the builds in my final system, so that I wont start from scratch again. Because of that, I had to constantly think about efficiency, testing and structure of my systems. In the next few sections, Im going to go more in detail about the compiler and the Abstract Machine which I had to develop at the start.

0.3 Simple Compiler

```
instance Monad CodeGen  where
  return val = MkCodeGen $ λnext → ([], next, val)
  ag >>= a2bg = MkCodeGen $ λnext →
    case codeGen ag next of
      (ac, next, a) → case
        codeGen (a2bg a) next of
          (bc, next, b) → (ac ++ bc, next, b)


genDef :: String → CodeGen Int
genDef code = MkCodeGen $ λnext → ([(next, code)], next + 1, next)


compile :: Expr → CodeGen Int
compile e = help "s" e  where
  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
```

0.4 Simple Abstract Machine

Basic Stack structure:

```
1      Mode: {
2          stack: { // rest of stack
3              prev: "", // previous stack
4              tag: "", // opperation
5              data: "" // pointer to next frame
6          }
7          tag: "",  // data type, example "num" for number
8          data: "" // value
9      }
```

Example program:

```
1      var ProgramFoo = [];
2
3  ProgramFoo[0] = function (s) {
4      return {
5          stack: s,
6          tag: "num",
7          data: 3
8      }
9  };
```

By passing this program to the Abstract Machine 3 will be placed on top of stack and then program machine will halt, since there are no further instructions.

Furthermore, lets examine how very simple Abstract Machine works.

Here we initialize starting variables, with mode.data being the most important bit, because it points to the starting instruction of the program that we pass in.

```
1      var save = [];
2      var mode = {
3          stack: null,
4          tag: "go",
5          data: 0
6      }
```

0.5 Project Evaluation

Project will be evaluated by other researchers who have a clear view of how the software should function. They will test the system and give their feedback and assessment. Evaluators will expect that the existing body of Frank examples should work in my implementation of the system the same way they do in other implementations, as well as client-side programming should

become easier, example, writing parsers for data in web forms. Furthermore, project will use continuous evaluation technique, so it will be evaluated, for example, every two weeks by at least one researcher, in order to follow Agile software development practices.

0.6 Progress Log

JANURAY

January 4th: Looked at Frankjr, Shonky, Vole implementations.

January 5th: Tried to install Frankjr for about 4 hours with no success, constant errors and crashes.

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

January 9th: Worked on project specification, plan and presentation. Also, tried bunch of new ways to install frank (cabal way) with no success.

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

January 11th: Worked on project specification, plan and presentation.

January 16th: Looked at Shonky stack implementation, tried to output it on the screen.

January 17th: Implemented simple linked list stack in JS, just as a pratice.

January 18th: Looked again at Shonky Abstract Machine. Understood the order how input program is parsed.

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

January 25th: Machine now works fine with stack larger than 2. It uses array to store functions now. Looking into how to implement throw, catch and compiler.

January 26th: Started to work on simple compiler written in Haskell.

January 30th: Simple compiler code compiles now. However, I still can't run it because I need to implement Show. My Haskell knowledge proves to be an issue. Also, laid the groundwork for latex with lex2tex, so I can start working on final report.

January 31th: Implemented Throw and Cache for Abstract Machine. Worked on final report.

FEBRUARY

February 01: Worked on latext configurations.

February 02: Implemented early versions of stack saving, restoring and variable assignment.

February 04: Report work - Introduction sections.