Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

Bachelor Thesis 2020

Ramon Rüttimann

# Functional Go

## Situation

With the rise of Javascript, Rust, and Go, the functional programming paradigm has gained popularity too. Though none of these programming languages are purely functional, they all share the common feature of having the possibility to use functional concepts.
However, a lot of programmers struggle initially with the concept of functional programming. Learning a purely functional programming language is extremely useful to gain familiarity with these concepts. Purely functional programming languages like Haskell though are not known for their beginner-friendliness. What makes learning a functional language difficult is that not only does the programmer have to learn an entirely different paradigm, but also a syntax that is uncommon for people coming from imperative or object-oriented languages.

## Objective

The objective is to ease the entry into functional programming. This should be achieved by providing the common functional toolset and an analysis tool for functional code in Go.

The common functional toolset, for example lists and operations on them, are found in pretty much every purely functional programming language. However, they are completely absent in Go. The difficulty here is that Go does not support polymorphism (as of Go 1), which means that most of these tools would have to be built into the compiler.

Go being a multi-paradigm language, it also does not enforce any specific programming style. Though this is preferred at normal usage of Go, it makes it hard for an unexperienced programmer to tell apart which concepts are purely functional and which are not.

To mitigate these points, the following work should be done:

- Implement a List-Type in Go's Compiler (similar to existing implementations of Slices and Maps)
- Implement commonly used functions that work upon lists, like `map`, `reduce`, etc.
- Write a code analysis tool that checks code and points out parts and constructs that are not purely functional

- Provide further linting for Sum Types[1] within that linter

If time allows it, further research with functional programming in Go could be conducted to find other issues that make it difficult to program in a purely functional style. These difficulties should then be made easier by either the code analysis tool or support within the compiler.

In the end, functional Go should still be syntactically familiar to people that have worked with Go (or C in that regard). With that, one can learn the concepts related to functional programming, without also needing to learn a new language and syntax.

## Submission Date

Friday, June 5th, 2020

Winterthur, February 10th, 2020

Gerrit Burkert, Karl Rege

## Update April 4th, 2020

New submission date is:
June 19th, 2020

[1] Sum types (sometimes also called a “tagged union” or variant”) are types that could take on one of several subtypes. They do not exist in Go (discussion on this can be found here: github.com/golang/go/issues/19412), though they are closely related to, and can be “implemented” with interfaces. However, on type switches, there is no possibility to do exhaustiveness checks, as the compiler does not know about sum types.