zh aw

School of Engineering

InIT Institute of Applied Information Technology

Bachelor thesis in Computer Science Spring 2020

Functional Go

...An Easier Introduction to Functional Programming

Author	Ramon Rüttimann
Main supervisor	Dr. G. Burkert
Sub supervisor	Dr. K. Rege
Date	06.06.2020

School of Engineering

DECLARATION OF ORIGINALITY

Bachelor's Thesis at the School of Engineering

By submitting this Bachelor's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date: Signature:

………………………………………………………….   ………………………………………………………………………..

                                                   ………………………………………………………………………..

                                                   ………………………………………………………………………..

The original signed and dated document (no copies) must be included after the title sheet in the ZHAW version of all Bachelor thesis submitted.

Summary

(in Deutsch)

Abstract

In the last decade, concepts from functional programming have grown in importance within the wider, non-functional programming community. Often it is recommended to learn a purely functional programming language such as Haskell to become familiar with these concepts. However, many programmers struggle with the double duty of learning a new paradigm and a new syntax at the same time. To lower this effort, this paper extends on the idea of learning purely functional programming with a multi-paradigm programming language and a familiar syntax.

For this task, the language of choice is Go due to its syntactical simplicity and familiarity. However, a downside of Go is the lack of a built-in list type, as lists take a central role in functional programming. Although this is remediated by Go's slices, they are not accompanied by any higher-order list processing functions — 'map', 'filter', and 'fold' to name a few — that are present in every functional programming language (and many other languages too). Due to the absence of polymorphism, in order to provide these higher-order functions in a user-friendly way it is necessary to build these functions into the compiler.

Furthermore, this paper settles on a definition of what pure functional programming is and introduces 'funcheck', a static code analysis tool that is designed to report constructs that are non-functional.

In conclusion, with the help of the newly built-in functions 'fmap', 'filter', 'foldr', 'foldl' and 'prepend', as well as 'funcheck' to lint code, Go proves itself to be a suitable language for getting started with functional programming. The primary factor for this is captured in the Go idiom 'clear is better than clever'. While functional Go code is more verbose when compared to functional languages, it is also more obvious about it's inner workings. At the same time, it also illustrates why there is no way around learning a language such as Haskell if fluency with functional programming concepts is desired. The main reasons are that, although it may be unusual at first, Haskell's syntax is extremely concise, and that the language's design — the type system, pattern matching, the purity guarantees and more — provides a very effective toolset for purely functional programming.

Preface

As a part of my bachelor studies, I chose to attend a course on functional programming. Having worked with Go for the last 3 years, first-class and higher-order functions were not particularly new ideas to me. However, learning Haskell was, at the beginning, overwhelming.

I decided to rewrite the exercises that I did not understand in Go. The result was more verbose; usually roughly two to three times the lines of code for the same algorithm. However, after writing the Go version, I understood not only the Go version, but also the Haskell version.

After doing this several times, I realised that I was constantly rewriting the same higher-order functions with different types, but more or less the same implementation. Thus, the idea of adding them as built-ins came up.

'Funcheck' then came into play when I wanted to build something in Go first and later rewrite it in Haskell. It was hard to tell whether it was purely functional, but it needed to be in order to be easier to write the implementation in Haskell.

This thesis is written based on my own struggles I had with Haskell, and it is my hope that someday, someone may benefit from the work done in this thesis.

Special thanks to:

Contents

1 Introduction

1.1 Learning Functional Programming

Within the last decade, concepts from functional programming have been brought into the daily life of almost every programmer. There are many events that contributed to this gain in popularity:

In 2007, C# 3.0 was released, which introduced lambda expressions and laid the foundations for turning C# into a hybrid Object-Oriented / Functional language[1]. Two years later, Ryan Dhal published the initial version of Node.js, eliminating JavaScript's ties to the browser and introducing it as a server-side programming language, increasing the adoption of JavaScript further. In 2013, Java 8 was released and brought support for lambda expressions and streams. Within the same time frame, Python has been rapidly growing in popularity[2].

Further, many new multi-paradigm programming languages have been introduced, including Rust, Kotlin, Go and Dart. They all have functions as first-class citizens in the language since their initial release.

With these developments it can be said that functional programming has truly landed in the wider programming-community, emerging from niche use-cases and academia. For example Rust, the 'most popular programming language' for 5 years in a row (2016–2020) according to the Stack Overflow Developer survey[3], has been significantly influenced by functional programming languages[4]. Further, in idiomatic Rust code, a functional style can be clearly observed[1].

Learning a purely functional programming language increases fluency with these concepts and teaches a different way to think and approach problems when programming. Due to this, many people recommend learning a functional programming language[5][6][7][8], even if one may not end up using that language at all[9].

Most literature about functional programming, including academia and online resources like blogs, contain code examples written in Haskell. Further, according to the Tiobe Index[10], Haskell is also the most popular purely functional programming language[11].

---

[1]A simple example for this may be that variables are immutable by default.

1.2 Haskell

Haskell, the *lingua franca* amongst functional programmers, is a lazily-evaluated, purely functional programming language. While Haskell's strengths stem from all it's features like type classes, type polymorphism, purity and more, these features are also what makes Haskell famously hard to learn[12][13][14][15].

Beginner Haskell programmers face a very distinctive challenge in contrast to learning a new, non-functional programming language: Not only do they need to learn a new language with an unusual syntax (compared to imperative or object-oriented languages), they also need to change their way of thinking and reasoning about problems. For example, the renowned quicksort-implementation from the Haskell Introduction Page[16]:

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Source Code 1.1: Quicksort implementation in Haskell

While this is only a very short and clean piece of code, these 6 lines already pose many challenges to non-experienced Haskellers:

– The function's signature with no 'fn' or 'func' statement as they often appear in imperative languages

– The pattern matching, which would be a 'switch' statement or a chain of 'if / else' conditions

– The deconstruction of the list within the pattern matching

– The functional nature of the program, passing '(< p)' (a function returning a function) to another function

– The function call to 'filter' without parenthesised arguments and no clear indicator at which arguments it takes and which types are returned

Although some of these constructs also exist in imperative or object-oriented languages, hte cumulative difference is not to underestimate and adds to Haskell's steep learning curve.

1.3 Goals

As demonstrated in the example above, learning a new paradigm and syntax at the same time can be daunting and discouraging for novices. The entry barrier for functional programming should be lowered by using a modern, multi-paradigm language with a clear and familiar syntax. The functional programming beginner should be able to focus on the paradigm first, and then change to a language like Haskell to fully get into functional programming.

To achieve this goal, this thesis will consist of two parts. In the first part, writing functional code will be made as easy as possible. This means that a programming language with an easy and familiar syntax should be chosen. Optimally, this language should already support functions as first-class citizens. Additionally, it should be statically typed, as a static type system makes it easier to reason about a program and can support the programmer while writing code. In the second part, a linter will be created to check code for non-functional statements. To achieve this, a definition of what functional purity means has to be selected and a ruleset has to be worked out and implemented into a static analysis tool.

1.4 Why Go

The language of choice for this task is Go, a statically typed, garbage-collected programming language designed at Google in 2009[17]. With its strong syntactic similarity to C, it should be familiar to most programmers.

Go strives for simplicity and its syntax is extremely small and easy to learn. For example, the language consists of only 25 keywords and purposefully omits constructs like the ternary operator (<bool> ? <then> : <else>) as a replacement for the longer 'if <bool> { <then> } else { <else> }' for clarity. 'A language needs only one conditional control flow construct'[18], and this also holds true for many other constructs. In Go, there is usually only one way to express something, improving the clarity of code.

Due to this clarity and unambiguity, the language is a perfect fit to grasp the concepts and trace the inner workings of functional programming. It should be easy to read code and understand what it does without a lot of experience with the language.

There are however a few downsides of using Go. Currently, Go does not have polymorphism, which means that functions always have to be written with specific types. Due to this, Go also does not include common list processing functions like 'map', 'filter', 'reduce' and

more[2]. Further, Go does not have a built-in 'list' datatype. However, Go's 'slices' cover a lot of use cases for lists already. Section 2.1 covers this topic in more detail.

1.5 Existing Work

With Go's support of some functional aspects, patterns and best practices have emerged that make us of functional programming constructs. For example, in the *net/http* package of the standard library, the function

```go
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

is used to register functions for http server handling[19]:

```go
func myHandler(w http.ResponseWriter, r *http.Request) {
    // Handle the given HTTP request
}

func main() {
    // register myHandler in the default ServeMux
    http.HandleFunc("/", myHandler)
    http.ListenAndServe(":8080", nil)
}
```

Using functions as function parameters or return types is a commonly used feature in Go, not just within the standard library. Furthermore, design patterns have emerged within the community that use functional concepts. An example of this are 'functional options'.

1.5.1 Functional Options

The 'functional options' pattern has been outlined in Dave Cheney's blog post 'Functional options for friendly APIs'[20] and is a great example on how to use the support for multiple paradigms. The basic idea with functional options is that a type constructor receives an unknown (0-n) amount of options:

---

[2]Although Go does have some polymorphic functions like 'append', these are specified as built-in functions in the language and not user-defined

```go
func New(requiredSetting string, opts ...option) *MyType {
  t := &MyType{
    setting: requiredSetting,
    featureX: false,
  }

  for _, opt := range opts {
    opt(t)
  }

  return t
}

type option func(t *MyType)
```

These options can then access the instance of `MyType` to modify it accordingly, for example:

```go
func EnableFeatureX() option {
  return func(t *MyType) {
    t.featureX = true
  }
}
```

To enable feature X, 'New' can be called with that option:

```go
t := New("required", EnableFeatureX())
```

With this pattern, it is easy to introduce new options without breaking old usages of the API. Furthermore, the typical 'config struct' pattern can be avoided and meaningful zero values can be set.

A more extensive example on how functional options are implemented and used can be found in Appendix 2.

In summary

    – *Functional options let you write APIs that can grow over time.*

   – *They enable the default use case to be the simplest.*

   – *They provide meaningful configuration parameters.*

   – *Finally they give you access to the entire power of the language to initialize complex values.*

   [20]

While this is a great example of what can be done with support for functional concepts, a purely functional approach to Go has so far been discouraged by the core Go team, which is understandable for a multi-paradigm programming language. However, multiple developers have already researched and tested Go's ability to do functional programming.

1.5.2 Functional Go?

In his talk 'Functional Go'[21], Francesc Campoy Flores analysed some commonly used functional language features in Haskell and how they can be copied to Go. Ignoring speed and stack overflows due to non-existent tail call optimisation[22], the main issue is with the type system and the missing polymorphism.

1.5.3 go-functional

In July 2017, Aaron Schlesinger, a Go programmer for Microsoft Azure, gave a talk on functional programming with Go. He released a repository[23] that contains 'core utilities for functional Programming in Go'. The project is currently unmaintained, but showcases functional programming concepts like currying, functors and monoids in Go. In the 'README' file of the repository, he also states that:

> *Note that the types herein are hard-coded for specific types, but you could use code generation to produce these FP constructs for any type you please! [24]*

1.6 Verdict

The aforementioned projects showcase the main issue with functional programming in Go: the missing helper functions that are prevalent in functional languages and that they currently cannot be implemented in a generic way.

To make functional programming more accessible in Go, this thesis will research what the most used higher-order functions are and implement them with a focus on usability. Furthermore, to learn purely functional programming, a list of rules for pure functional

code should be curated and implemented in a static code analysis tool. This tool can then be used to check existing code and report constructs that are not functional.

2 About Go

This chapter introduces the core concepts in Go that are needed to follow this paper. Go is a language similar to C, although with a few minor, but important differences. First, Go is garbage collected, meaning that the programmer does not need to allocate and free memory[1]. Secondly, Go does not allow for pointer arithmetic. 'Without pointer arithmetic it's possible to create a language that can never derive an illegal address that succeeds incorrectly'[25].

Further, Go provides a built-in data type that does not exist in plain C: slices.

2.1 Go Slices

As mentioned in Section 1.4, Go does not have a 'list' implementation and lists are rarely used. The reason for this is twofold. First, as Go does not have polymorphism, it is not possible for users to implement a generic 'list' type that would work with any underlying type. Second, the Go authors added 'slices' as a core type to the language. From a usage perspective, lists would not add anything compared to slices.

Go's Slices can be viewed as an abstraction over arrays, to mitigate some of the weaknesses of arrays when compared to lists.

> Arrays have their place, but they're a bit inflexible, so you don't see them too often in Go code. Slices, though, are everywhere. They build on arrays to provide great power and convenience.[26]

Slices can be visualised as a 'struct' over an array:

```go
// NOTE: this type does not really exist, it
// is just to visualise how they are implemented.
type Slice struct {
    // the underlying "backing store" array
    array *[]T
```

---

[1]although allocating memory is possible with the `new` built-in function

```
    // the length of the slice / view on the array
    len int
    // the capacity of the array from the
    // starting index of the slice
    cap int
}
```

With the 'append' function, elements can be added to a slice. Should the underlying array not have enough capacity left to store the new elements, a new array will be created and the data from the old array will be copied into the new one. This happens transparently to the user.

2.1.1 Using Slices

'head', 'tail' and 'last' operations can be done with index expressions:

```
// []<T> initialises a slice, while [n]<T> initialises an
// array, which is of fixed length n. One can also use `...'
// instead of a natural number, to let the compiler count
// the number of elements.
s := []string{"first", "second", "third"}
head := s[0]
tail := s[1:]
last := s[len(s)-1]
```

Adding elements or joining slices is achieved with 'append':

```
s := []string{"first", "second"}
s = append(s, "third", "fourth")
t := []string{"fifth", "seventh"}
s = append(s, t...)
// to prepend an element, one has to create a
// slice out of that element
s = append([]string{"zeroth"}, s...)
```

Append is a variadic function, meaning it takes n elements. If the slice is of type $[]<T>$, the appended elements have to be of type $<T>$.

12

To join two lists, the second list is expanded into variadic arguments.

More complex operations like removing elements, inserting elements in the middle or finding elements in a slice require helper functions, which have also been documented in Go's Slice Tricks[27].

2.1.2 What is missing from Slices

This quick glance at slices should clarify that, though the runtime characteristics of lists and slices can differ, from a usage standpoint, what is possible with lists is also possible with slices.

In a typical program written in a functional language, lists take a central role[2]. Because of this, functional languages have a number of helper functions like 'map', 'filter' and 'fold'[29] to modify and work on lists. These so called 'higher order functions' currently do not exist in Go and would need to be implemented by the programmer. With no support for polymorphism, a different implementation would need to be written for every slice-type that is used. The type []int (read: a slice of integers) differs from []string, which means that a possible 'map' function would have to be written once to support slices of integers, once to support slices of strings, and a combination of these two:

```go
func mapIntToInt(f func(int) int, []int) []int
func mapIntToString(f func(int) string, []int) []string
func mapStringToInt(f func(string) int, []string) []int
func mapStringToString(f func(string) string, []string) []string
```

With 7 base types (eliding the different 'int' types like 'int8', 'uint16', 'int16', etc.), this would mean $7^2 = 49$ map functions just to cover the base types. Counting the different numeric types into that equation (totally 19 distinct types[30]), would grow that number to $19^2 = 361$ functions.

Though this code could be generated, it misses user-defined types which would still need to be generated separately in a pre-compile step.

Another option, instead of having a function per type, would be that 'map' takes and returns empty interfaces (interface{}). However, 'the empty interface says nothing'[31]. The declaration of 'map' would be:

---

[2]Interestingly, there is no clear and direct answer as to why they do. One reason may be because they are recursively defined and trivially to implement functionally. Further, they are easier to use than arrays, where the programmer would need to track the index and bound of the array (imagine keeping track of the indices in a recursive function)[28]

```go
func map(f func(interface{}) interface{}, interface{}) interface{}
```

This function header does not say anything about it's types, which would mean that they would need to be checked at runtime and handled gracefully. It would also require the caller to do a type assertion after every call. Further, a slice of type `T` cannot simply be converted or asserted to a slice of another type `T2`[32][33]. Because of this limitation, a typical usage pattern of map would be:

```go
s := []string{"hello", "world"}
var i []interface{}
for _, e := range s {
  i = append(i, e)
}
// i is now populated and can be used.
r := map(someFunc, f)
// to convert it back to []string:
s = r.([]string)
```

This exemplifies why using the empty interface is not an option. Further, the function could not really be type-checked at compile type, as there is no indication of which argument's type must be equal.

2.2 Built-in functions

To mitigate these issues, the most common list-operations (in Go slice-operations) will be added as built-ins to the compiler, so that the programmer can use these functions on every slice-type without any conversion or code generation being necessary.

The language specification defines what built-in functions are and which built-in functions should exist:

> *Built-in functions are predeclared. They are called like any other function but some of them accept a type instead of an expression as the first argument.*
>
> *The built-in functions do not have standard Go types, so they can only appear in call expressions; they cannot be used as function values.[34]*

For example, the documentation for the built-in `append`:

```
// The append built-in function appends elements to the end of a slice.
// ...
func append(slice []Type, elems ...Type) []Type
```

The documentation shows that the types supplied to append are not specified upfront. Instead, they are resolved and checked during compilation of the program.

Thus, in order to have generic list processing functions, these functions need to be implemented as built-ins in the compiler.

2.3 The Go Compiler

The Go programming language is defined by its specification[35], and not it's implementation. As of Go 1.14, there are two major implementations of that specification; Google's self-hosting compiler toolchain 'gc', which is written in Go, and 'gccgo', a front end for GCC, written in C++.

When talking about the Go compiler, what's mostly referred to is 'gc'[3].

A famous, although not completely correct story tells about Go being designed while a C++ program was compiling[36]. This is why one of the main goals when designing Go was fast compilation times:

> Finally, working with Go is intended to be fast: it should take at most a few seconds to build a large executable on a single computer. To meet these goals required addressing a number of linguistic issues: an expressive but lightweight type system; concurrency and garbage collection; rigid dependency specification; and so on. These cannot be addressed well by libraries or tools; a new language was called for.[37]

Go has taken some measures to combat slow compilation times. In general, Go's dependency resolution is simpler compared to other languages, for example by not allowing circular dependencies. Furthermore, compilation is not even attempted if there are unused imports or unused declarations of variables, types and functions. This leads to less code to compile and in turn shorter compilation times. Another reason is that 'the 'gc' compiler is simpler code compared to most recent compilers'[38]. However, according to Rob Pike, one of the creators of Go, Go's compiler is not notably fast, but most other compilers are slow:

---

[3]'gc' stands for 'go compiler', and not 'garbage collection' (which is abbreviated as 'GC').

> *The compiler hasn't even been properly tuned for speed. A truly fast compiler*
> *would generate the same quality code much faster.[38]*

The code generation with the 'gc' compiler is split into four phases:

2.3.1 Parsing Go programs

The first phase of compilation is parsing Go programs into a syntax tree. This is done by tokenising the code ('lexical analysis' - the 'lexer'), parsing ('syntax analysis' - the 'parser') it and then constructing a syntax tree (AST)[4] for each source file.

Go has been designed to be easy to parse and analyse. For example, in contrast to many other programming languages, Go does not require a symbol table to parse source code.[40].

2.3.2 Type-checking and AST-transformation

The second phase of compilation starts by converting the 'syntax' package's AST, created in the first phase, to the compiler's AST representation. This is due to historical reasons, as gc's AST definition was carried over from the C implementation.

After the conversion, the AST is type-checked. Within the type-checking, there are also some additional steps included like checking for 'declared and not used' variables and determining whether a function terminates.

After type-checking, transformations are applied on the AST. This includes eliminating dead code, inlining function calls and escape analysis, to name a few. What is also done in the transformation phase is rewriting built-in function calls, replacing for example a call to the built-in 'append' with the necessary AST structure and runtime-calls to implement its functionality.

2.3.3 SSA

In the third phase, the AST is converted to SSA[5] form. SSA is 'a lower-level intermediate representation with specific properties that make it easier to implement optimizations and to eventually generate machine code from it'[41].

---

[4]Technically, the syntax tree is a syntax DAG[39]
[5]Single Static Assignment, an intermediate representation between the AST and the compiled binary that simplifies and improves compiler optimisations

The conversion consists of multiple 'passes' through the SSA that apply machine-independent rules to optimise code. These generic rewrite rules are applied on every architecture and thus mostly concern expressions (for example replacing expressions with constant values and optimising multiplications), dead code elimination and removal of unneeded nil-checks.

2.3.4 Generating machine code

Lastly, in the fourth phase of the compilation, machine-specific SSA optimisations are applied. These may include:

– Rewriting generic values into their machine-specific variants (for example, on amd64, combining load-store operations)

– Dead-code elimination

– Pointer liveness analysis

– Removing unused local variables

After generating and optimising the SSA form, the code is passed to the assembler, which replaces the so far generic instructions with architecture-specific machine code and writes out the final object file[41].

3 Methodology

3.1 Slice Helper Functions

3.1.1 Choosing the functions

As mentioned in the introduction, the goal is to make functional programming in Go easier. To achieve this, the first task is to implement some helper functions for slices, similar to those that exist for lists in Haskell. To decide on which functions will be implemented, popular Haskell repositories on GitHub have been analysed. The popularity of repositories was decided to be based on their number of stars. Out of all Haskell projects on GitHub, the most popular are[42]:

- Shellcheck (koalaman/shellcheck[43]): A static analysis tool for shell scripts
- Pandoc (jgm/pandoc[44]): A universal markup converter
- Postgrest (PostgREST/postgrest[45]): REST API for any Postgres database
- Semantic (github/semantic[46]): Parsing, analyzing, and comparing source code across many languages
- Purescript (purescript/purescript[47]): A strongly-typed language that compiles to JavaScript
- Compiler (elm/compiler[48]): Compiler for Elm, a functional language for reliable web apps
- Haxl (facebook/haxl[49]): A Haskell library that simplifies access to remote data, such as databases or web-based services

In these repositories, the number of occurrences of popular list functions has been counted. The analysis does not differentiate between different 'kinds' of functions. For example, 'fold' includes all occurrences of `foldr`, `foldl` and `foldl'`; 'map' includes occurrences of `fmap`, `mapM` and `mapM\_`. Also, the analysis has not been done with any kind of AST-parsing. Rather, a simple 'grep' has been used to find matches. This means that it is likely to contain some mismatches, for example in code comments. All in all, this analysis should only be an indicator of what functions are used most.

Running the analysis on the 7 repositories listed above, searching for a number of pre-selected list functions, indicates that the most used functions are ':' (cons), 'map' and 'fold', as shown in table 3.1.

':' (cons)	2912
map	1241
fold	610
filter	562
take	376
drop	238
reverse	185
sum	173
zip	158
product	56
maximum	53
minimum	26

Table 3.1: Occurrences of list functions[1]

Based on this information, it has been decided to implement the map, cons, fold and filter functions into the Go compiler[2].

3.1.2 Map

The most used function in Haskell is map. The table 3.1 counts roughly 1250 occurrences of map, although around 600 of those are from 'fmap'. fmap is part of the Functor type class[3], which is described as 'a type that can be mapped over'[50]. In general, a common analogy of the Functor type class is a box. A functor is like a box where a value can be put into and taken out again. fmap processes and transforms the item in that box. For lists, this process means iterating over and processing every item within that list. For the

---

[1]See Appendix 3 for more information

[2]The cons function (':') is overrepresented in this list, as it can also be used to deconstruct lists within pattern matching. Filtering these occurrences would need a more advanced algorithm. However, even if $\frac{3}{4}$ of these usages are used in pattern matching, the cons function would still be in the top 3.

[3]type classes in Haskell are similar to interfaces in imperative and object-oriented languages

'Maybe' type it means 'unpacking' the concrete value and processing it, or if there is no concrete value, returning 'Nothing' instead.

The map function is exactly like fmap but only works on lists:

$$map$$

> returns a list constructed by applying a function (the first argument) to all items in a list passed as the second argument[51].

Some usage examples of map can be seen at 3.1.

```
Prelude> :t map
map :: (a -> b) -> [a] -> [b]
Prelude> :t fmap
fmap :: Functor f => (a -> b) -> f a -> f b
Prelude> map (*3) [1,2,3]
[3,6,9]
Prelude> fmap (*3) [1,2,3]
[3,6,9]
Prelude> map (++ " world") ["hello","goodbye"]
["hello world","goodbye world"]
Prelude> map show [1,2,3]
["1","2","3"]
```

Source Code 3.1: Example usage for map and fmap

Due to missing polymorphism, map cannot be implemented easily in Go. While a specific definition of map would be

```
func map(f func(int) string, []int) []string
```

this definition would only hold true for the specific type combination `int` and `string`. A more generic definition, similar to append, would be:

```
func map(f func(Type1) Type2, []Type1) []Type2
```

For this to work, the function has to be implemented as a built-in into the compiler.

As there is already a 'map' token in the Go compiler (for the map data type), the function will be called 'fmap'. However, compared to Haskell's 'fmap', Go's 'fmap' only works on slices. This is due to the absence of an 'functor'-like concept. Again, due to the absence of polymorphism, this cannot realistically built into the language in this context.

Nonetheless, to avoid possible naming confusions, the 'map' function in Go will be called 'fmap'.

In Go, the usage of 'fmap' should result in making the program 3.2 behave as shown[4].

```go
package main

import (
  "fmt"
  "strconv"
)

func main() {
  fmt.Printf("%#v", fmap(strconv.Itoa, []int{1, 2, 3}))
  ↪ // []string{"1", "2", "3"}
}
```

Source Code 3.2: Example usage of map in Go

3.1.3 Cons

The name cons has been introduced by LISP, where it describes a record structure containing two components called the 'car' (the '**c**ontents of the **a**ddress **r**egister') and the 'cdr' ('**c**ontent of **d**ecrement **r**egister'). Lists are built upon cons cells, where the 'car' stores the element and 'cdr' a pointer to the next cell - the next element of the list. This is why in Lisp, (cons 1 (cons 2 (cons 3 (cons 4 nil)))) is equal to (list 1 2 3 4). This list is also visualised in picture 3.1.

The cons operator thus prepends an element to a list, effectively allocating a variable that contains the newly added element and a pointer to the 'old' list. As a result, prepending to a list is computationally cheap, needing one allocation and one update.

In Haskell, the 'name' of the cons function is the ':' operator. In Go, names for identifiers (which includes function names) underlie a simple rule:

---

[4]Printf's first argument, the verb '%#v', can be used to print the type ('#') and the value ('v') of a variable[52].
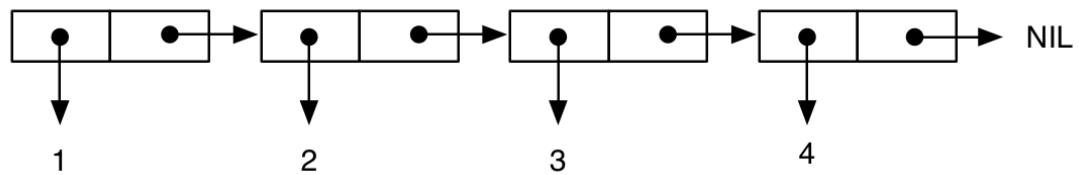
Figure 3.1: Cons cells forming a list[53]

> *An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.[54]*

This rule forbids a function to be named ':'. Instead, the function could be named 'cons'. However, Go already has a function to add to the end of a slice, 'append'. Thus, adding to the beginning of a slice will be named 'prepend'. Using prepend should be very similar to append. The behaviour of 'prepend' should be equal to using 'append' with a slight workaround[5]:

```go
package main

import (
  "fmt"
  "strconv"
)

func main() {
  fmt.Printf("%#v", append([]int{0},
  ↪  []int{1,2,3}...)// []int{0, 1, 2, 3}
  fmt.Printf("%#v", prepend(0, []int{1, 2, 3})) // []int{0, 1, 2, 3}
}
```

Source Code 3.3: Example usage of prepend in go

3.1.4 Fold

Fold, sometimes also named 'reduce' or 'aggregate', is another higher-order function that is very commonly used in functional programming.

---

[5]This workaround is to create a slice containing the prepended element, and expanding the destination slice with '...', as `append` is a variadic function

> *fold refers to a family of higher-order functions that analyze a recursive data*
> *structure and through use of a given combining operation, recombine the results*
> *of recursively processing its constituent parts, building up a return value.[55]*

In other words, fold processes a list one by one and executes a 'combining operation' on every element, for example summing up a list of integers.

The family of fold functions in Haskell consist of three different implementations of that definition: foldr, foldl and foldl'. The difference between foldr and foldl is hinted at their function headers:

```
Prelude Data.List> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude Data.List> :t foldr
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
```

Source Code 3.4: Function headers of the fold functions

The argument with type 'b' is passed as the first argument to the foldl function, and as the second argument to foldr. As can be seen in the illustrations of foldl and foldr in 3.2, the evaluation order of the two functions differ.
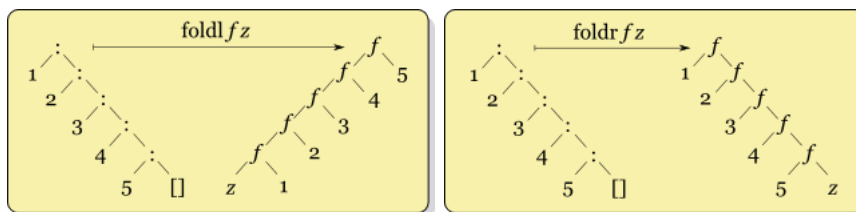

Figure 3.2: Folds illustrated[55]

This is most obvious when using an example where the function is not associative:

```
foldl (-) 0 [1..7]
((((((0 - 1) - 2) - 3) - 4) - 5) - 6) - 7 = -28
foldr (-) 0 [1..7]
1 - (2 - (3 - (4 - (5 - (6 - (7 - 0)))))) = 4
```

Source Code 3.5: foldr and foldl execution order

In foldl, the accumulator ('0') is added to the left end of the list (prepended), while with foldr, the accumulator is added to the right end. For associative functions (e.g. '+') this does not make a difference, it does however for non-associative functions, as can be seen in the example 3.5.

The difference between foldl and foldl' is more subtle:

> *foldl and foldl' are the same except for their strictness properties, so if both return a result, it must be the same.[56]*

The strictness property is only relevant if the function is lazy in its first argument. If this is the case, behavioural differences can be seen because foldl builds up a so called 'execution path' (nesting the called functions), while foldl' executes these functions while traversing it already. An example of this is illustrated in Appendix 6.

To keep things simpler, Go will only have its versions of foldl and foldr, which will both be strict — the Haskell counterparts would thus be foldr and foldl'.[6] The usage of these fold-functions is equal to the Haskell versions, where foldl's arguments are switched in order.

```go
package main

import "fmt"

func main() {
  sub := func(x, y int) int { return x - y }
  fmt.Printf("%v\n", foldr(sub, 100, []int{10, 20, 30})) // -80
  fmt.Printf("%v\n", foldl(sub, 100, []int{10, 20, 30})) // 40
}
```

Source Code 3.6: Example usage of foldr and foldl in go

3.1.5 Filter

The filter function is the conceptually simplest higher-order function. It takes a list and filters out all elements that are not matching a given predicate. This predicate usually is a function that takes said element and returns a boolean if it should be kept or filtered.

A simple example:

---

[6]If the behaviour from the normal foldl function is required, a workaround can be applied in the Go version. See Appendix 7

```go
package main

import "fmt"

func main() {
  smallerThan5 := func(x int) bool {
    return x < 5
  }

  fmt.Println(filter(smallerThan5, []int{1, 8, 5, 4, 7, 3}))
  ↪  // [1, 4, 3]
}
```

Source Code 3.7: Example usage of filter in Go

3.2 Functional Check

To learn functional programming without a purely functional language, the developer needs to know which statements are functional and which would not exist or be possible in a purely functional language. For this reason, a 'functional checker' should be created. It should work in a similar fashion to linters like 'shellcheck', 'go vet' or 'gosimple'[7], with different 'rules' that are reported upon. This tool will be named 'funcheck'.

A set of rules should be compiled to identify common 'non-functional' constructs which should then be reported upon.

The first step is to define a set of rules.

3.2.1 Functional Purity

The goal of funcheck is to report every construct that can not be considered to be purely functional. As there is no agreed upon definition on what functional purity is[58], the first step is to define functional purity for the context of this thesis:

> *A style of building the structure and elements of computer programs that treats all computation as the evaluation of mathematical functions. Purely functional*

---

[7]A list of Go linters can be found in golangci-lint[57].

programming may also be defined by forbidding changing state and mutable data.

Purely functional programming consists in ensuring that functions, inside the functional paradigm, will only depend on their arguments, regardless of any global or local state.[59]

This definition roughly brakes down into two parts; immutability and function purity. These parts and how they translate to Go will be discussed in the following chapters.

3.2.2 Forbidding mutability and changing state

Immutability refers to objects — variables — that are unchangeable. This means that their state cannot be modified after it's creation and first assignment.

Many programming languages provide features for making variables immutable.

In Rust for example, variables are immutable by default. To explicitly declare a mutable variable, the `mut` keyword has to be used: `let mut x = 5;`.

Java, in contrast, uses the `final` keyword:

A final variable can only be initialized once[60]

'Only be initialized once' means that it cannot be reassigned later, effectively making that variable immutable. A caveat with Java's `final` keyword is that the immutability is only tied to the reference of that object, not it's contents (one may still change a field of an immutable object).

C and C++ have two features to achieve immutability, the `#define` preprocessor and the `const` keyword. The `#define` directive does not declare variables, but rather works in a similar way to string replacement at compile time. These directives can also be expressions. For example, this is a valid define statement:

```
#define AGE (20/2)
```

However, because `#define` is just text substitution, these identifiers are untyped.

The `const` qualifier specifies that a variable's value will not be changed, making it immutable. This is effectively the equivalent to Java's `final`.

Go has the `const` keyword too, and similar to C, constants can only be characters, strings, booleans or numeric values[8]. A complex type — a struct, interface or function — cannot be constant.

This means that the programmer cannot write `const x = MyStruct{A: a, B: b}` to make a struct immutable.

A solution to making variables immutable is to explicitly not allow any mutations in a program, effectively disallowing all reassignments. To do this in Go, the simplest solution is to disallow the assignment operator `=`, and only allow it in combination with a declaration (`:=`).

This solution also has the side-effect that it makes it impossible to mutate existing, complex data structures like maps and slices[9].

There are additional operators that work in a similar way to the assignment operator, for example `++`, `--` and `+=`. These are all 'hidden' assignments and will need to be reported upon too.

Go being a copy by value language, mutating existing variables across functions works by passing pointers to these variables[10]. When removing the regular assignment operators, the usage of pointers becomes second nature, as it cannot be updated either way. Technically, the only advantage that pointers have left, is that less memory has to be copied[11] if a variable's type is big enough[12]. This optimisation is left to the developer.

In functional languages in contrast, this optimisation is usually left to the compiler and there are no pointers at all in the language[13].

A feature not discussed so far is shadowing. Shadowing a variable in a different scope is still possible by redeclaring it. As expected, even with the above mentioned limitations, the rules for shadowing variables do not change. This can be seen in Appendix 5.

3.2.3 Pure functions

A pure function is a function where the return value is only dependent on the arguments. With the same arguments, the function should always return the same values. The

---

[8]In contrast to C, Go does have a boolean type

[9]By not allowing assignments, elements can not be updated by `s[0] = "zero"` / `m["key"] = "value"`.

[10]An example for this can be found in Appendix 4.

[11]The pointer still has to be passed, and thus copied, to the called function

[12]For example, it is cheaper to copy an `int32` than to take address of it, copy that (64 bit, on most systems) address and dereferencing it again

[13]Except for things like the 'foreign function interface' (FFI) in Haskell, which allows to call C functions from Haskell code

evaluation of the function is also not allowed to have any 'side effects', meaning that it should not mutate non-local variables or references[14].

As discussed in the last Chapter 3.2.2, if reassignments are not allowed, mutating global variables is not possible either.

If global state cannot be changed, it also cannot influence a function's return values.

As such, forbidding reassignments is an extremely simple solution to ensure functional purity within the program. However, there is one important aspect that the solution does not solve: interacting with the outside world.

3.2.4 IO

Network and file access, time, randomness, user input and sensors are all examples of IO, things that interact with state outside of the program.

Input and Output in a program — through whether channel it may be — is impure by design. Due to this, the Haskell language authors faced a difficult challenge; how to add impure functions to an otherwise completely pure language.

In a pure language, functions only depend on their arguments, have no side effects and return a value computed from these arguments. Because of this, the execution order of functions is not relevant, a function can be executed whenever its return value is needed. Due to this, the compiler can optimise the code, calling the functions in a specific (or in no specific) order or omitting calls if their return values are unused.

However, impure IO functions introduce some difficulties.

For example, one would expect that it is possible to write a trivial 'writeFile' function that takes a filename and a string with the contents for that file with the function definition roughly being:

```
writeFile :: String -> String
```

Supposed that this function could be called with `writeFile "hello.txt" "Hello World"`. The function does not return anything, or its return value, for example the number of bytes that have been written to the file, is discarded on the call site.

As there is no dependency on that function anymore, the compiler assumes that this function does not need to be executed at all — it still expects all functions to be pure

---

[14]This also includes 'local static' variables, but Go does not have a way to make variables 'static'.

(even if this is not the case in this example). Being able to follow the purity guarantees, the compiler emits the call to the writeFile function and the file never gets written.

Similarly, for getting user input:

```
getChar :: Char
```

If this function is called multiple times, the compiler may reorder the execution of these calls, again, because it expects the functions to be pure, so the execution order should not matter:

```
get2Chars = [getChar, getChar]
```

However, in this example, execution order is extremely important.

To solve this problem, Haskell introduced the IO monad, in which all functions are wrapped with a 'RealWorld' argument. So the getChar example would be written as[15]:

```
getChar :: RealWorld -> (Char, RealWorld)
```

This can be read as 'take the current RealWorld, do something with it, and return a Char and a (possibly changed) RealWorld'[61].

With this solution, the compiler cannot reorder or omit IO actions, ensuring the correct execution of the program.

The IO monad is a sophisticated solution to work around the purity guarantees and enables Haskellers to use impure functions in an otherwise completely pure language. Put another way: the IO monad exists because of compiler optimisations, which in turn are based on the assumption that everything is pure.

In Go, the compiler does not and can not make this assumption. There also is no 'pure' keyword to mark a function as such. For that reason, the compiler cannot optimise as aggressivly, and the whole IO problem does not exist. However, in the context of this thesis, interacting with the outside world — using IO-functions — means that functional purity cannot be guaranteed anymore. Haskell's solution to this issue is relatively complex, and while it may be applicable to Go, it would require massively rewriting Go's standard library. Furthermore, the IO monad does not make impure functions magically pure. Rather, it clearly marks functions as impure.

---

[15] This is not the actual function header for getChar, only an illustration. The actual function header of getChar is `getChar :: IO Char`.

Another issue, apart from the amount of work that would be needed, is that the goal is to provide an easy introduction to functional programming. IO, as one may see in these examples, is one of the hardest topics.

For those reasons, although it may violate the functional purity requirements, the underlying issue with IO will be ignored. This means that the programmer will still be able to use IO functions as in regular Go programs.

3.2.5 Assignments in Go

As described in Section 3.2.2, the only check that needs to be done is that there are no reassignments in the code. There are a few exceptions to this rule, which will be covered later. First, it is important to have a few examples and test cases to be clear on what should be reported.

To declare variables in Go, the `var` keyword is used. The var keyword needs to be followed by one or more identifiers and maybe types or the initial value. These are all valid declarations:

```go
var i int
var U, V, W float64
var k = 1
var x, y float32 = -1, -2
```

Source Code 3.8: Go Variable Declarations

What can be seen in this example is that the type of the variable can be deducted automatically[16], or be specified explicitly.

However, the notation `var k = 1` is seldomly seen. This is due to the fact that there is a second way to declare and initialise variables to a specific value, the 'short variable declaration' syntax:

```go
k := 1
```

---

[16]The compiler will infer the type at compile time. That operation is always successful, although it may not be what the programmer desires. For example, `var x = 5` will always result in `x` to be of type `int`.

It is shorthand for a regular variable declaration with initializer expressions but no types:

```
"var" IdentifierList = ExpressionList .
```

[62]

In practice, the short variable declaration is used more often due to the fact that there is no need to specify the type of the variable manually and is shorter than its counterpart `var x = y`.

Go also has increment, decrement and various other operators to reassign variables. Most of the operators that take a 'left' and 'right' expression also have a short-hand notation to reassign the value back to the variable, for example[17]:

```
var x = 5
x += 5 // equal to x = x + 5
x %= 3 // equal to x = x % 3
x <<= 2 // equal to x = x << 2 (bit-shift)
```

Source Code 3.9: Go Assignment Operators

All these operators are reassigning values and should be reported.

However, as mentioned in the beginning of this chapter, there are a few exceptions to this rule:

Function Assignments

A variable declaration brings an identifier into scope. The Go Language Specification defines this scope according to the following rule:

> *The scope of a constant or variable identifier declared inside a function begins at the end of the ConstSpec or VarSpec (ShortVarDecl for short variable declarations) and ends at the end of the innermost containing block. [64]*

This definition holds an important caveat for function definitions:

> *The scope [...] begins **at the end** of the ConstSpec or VarSpec [...].*

---

[17]This is a non-exhaustive list of operators. A complete listing of operators can be found in the Go language spec[63].

This means that when defining a function, the functions identifier is not in scope within that function:

```go
func X() {
  f := func() {
    f() // <- 'f' is not in scope yet.
  }
}
```

Source Code 3.10: Go scoping issue with recursive functions

In the above example, the function `f` cannot be called recursively. This is due to the fact that the identifier `f` is not in scope at `f`'s definition. Instead, it only enters scope after the closing brace of function `f`.

This cannot be changed within the compiler without touching a lot of the scoping rules and changing these scoping rules may have unintended side effects. To exemplify, a naive solution to the issue would be to bring the identifier into scope right after the assignment operator (`=`). However, this would introduce further edge cases, as for example `x := x`, and extra measures would have to be taken to make such constructs illegal again.

However, there is a simple, although verbose solution to this problem: The function has to be declared in advance:

```go
func X() {
  var f func() // f enters scope after this line
  f = func() {
    f() // this is allowed, as f is in scope
  }
}
```

Source Code 3.11: Fixing the scope issue on recursive functions

This exception will need to be allowed by the assignment checker, as recursive functions are one of the building blocks in functional programming.

Multiple Assignments

As discussed at the beginning of this Chapter, the short variable declaration brings a variable into scope and assigns it to a given expression (value). It is also possible to declare multiple variables at once, as can be seen in the EBNF notation:

```
ShortVarDecl = IdentifierList ":=" ExpressionList .
```

This clearly shows that both left- and right-hand side take a list, meaning one or more elements, making a statement like x, y := 1, 2 valid[18].

However, there is an important note to be found in the language specification:

> Unlike regular variable declarations, a short variable declaration may redeclare variables provided they were originally declared earlier in the same block (or the parameter lists if the block is the function body) with the same type, and at least one of the non-blank[19] variables is new. As a consequence, redeclaration can only appear in a multi-variable short declaration. Redeclaration does not introduce a new variable; it **just assigns a new value to the original**[62].

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)  // redeclares offset
```

This should not be allowed, as the programmer can introduce mutation by redeclaring already existing variables in a short variable declaration.

For Loops

Although the aforementioned rules cover almost every case of reassignment, they miss one of the most important parts: For loops.

In Go, there is only one looping construct, the `for` loop. However, it can be used in roughly four different ways:

The infinite loop:

---

[18]This also holds true for the `var` keyword

[19]The blank identifier (_) discards a value; x, _ := f()

```go
for {
  // ...
}
```

The while loop:

```go
for x == true { // boolean expression
  // ...
}
```

The regular C-style for loop:

```go
for x := 0; x < 10; x++ {
  // ...
}
```

And the range loop, which allows to range (iterate) over slices, maps and channels[20]:

```go
for idx, elem := range []int{1,8,5,4} {
  // ...
}
```

Range returns two values, both of which can be omitted[21]. When ranging over a slice, the index and a copy of the element at that index is returned. For maps, the key and a copy of the corresponding value is returned.

Except of the C-style for loop, the loops do not have a reassignment statement when converted to the AST. For this reason, they would not be detected by the previously defined rule and thus the programmer could use them. However, issues emerge when having a closer look at the loop constructs.

The `range` loop needs to keep track of the element, and it does so (internally) by using a pointer. This results in a (internal) reassignment.

---

[20] Channels have not been discussed in this thesis, but are an important building block in Go's concurrency features. They work similarly to Unix Pipes ('|'), in that they allow to send data in and receive it on the other 'side'. A `range` receives as long as the channel is not closed.

[21] The values can be omitted by using the blank identifier '`_`'. Skipping the index is thus `_, elem := range s`, while skipping the value with `i, _ := range s` which is equal to just `i := range s`.

The regular C-style for loop and the 'while' loop do not work at all without reassignments. Although in the while-loop the reassignment happens at another place, most probably within the loop, it is easier for the user if a report on the loop construct is generated instead.

The infinite loop does not contain any reassignments and is thereby be legal by the definition of the reassignment rule. Because of this, it should not be reported.

All other for-loops need to be reported as they contain internal reassignments.

4 Implementation

4.1 Implementing the new built-in functions

4.1.1 Required Steps

Adding a built-in function to the Go language requires a few more steps than just adding support within the compiler. While it would technically be enough to support the translation between Go code and the compiled binary, there would be no visibility for a developer that there is a function that could be used. For a complete implementation, the following steps are necessary:

- Adding the Godoc[65] that describes the function and it's usage

- Adding type-checking support in external packages for tools like Gopls[1]

- Adding the implementation within the internal[2] package of the compiler

    - Adding the AST node type

    - Adding type-checking for that node type

    - Adding the AST traversal ('walk') for that node type, translating it to AST nodes that the compiler already knows and can translate to built-in runtime-calls or SSA

The Go source code that is relevant for this thesis can be classified into three different types. One is the Godoc — the documentation for the new built-in functions. The other two are the 'public' and the 'private' implementation of these built-ins.

The 'private' implementation is located within the *src/cmd/compile/internal* package[67]. Because it is an internal package, it can only be used by the packages in *src/cmd/compile*, which contain the implementation of the compiler itself.

When calling

---

[1]Gopls is Go's official language server implementation[66].

[2]'internal' packages can only be imported by other packages that are rooted at the parent of the 'internal' directory. It is used to make specific packages not importable in order to decrease potential API surface[67].

```
$> go build .
```

the compiler is invoked indirectly through the main 'go' binary. To directly invoke the compiler,

```
$> go tool compile
```

can be used.

Everything that is not in *src/cmd/compile* is referred to as the 'public' part of the compiler in this thesis. The 'public' parts are used by external tools, for example Gopls, for type-checking, source code validation and analysis.

4.1.2 Adding the Godoc

In Go, documentation is generated directly from comments within the source code [65]. This also applies to built-in functions in the compiler, which have a function stub to document their behaviour[68], but no implementation, as that is done in the compiler[69].

The documentation for built-ins should be as short and precise as possible. The usage of 'Type' and 'Type1' has been decided based on other built-ins like 'append' and 'delete'. The function headers are derived from their Haskell counterparts, adjusted to the Go nomenclature.

```
135   // ...
      // The prepend built-in function prepends an element to the start of
      // the slice. This function always returns a copy of the underlying
      // array, with the given element at the head of that array. It is
      // therefore necessary to store the result of prepend, often in the
140   // variable holding the slice itself.
      func prepend(elem Type, slice []Type) []Type

      // The fmap built-in function maps a slice of elements from one type to
      // a slice of elements of another type, using the given transformation
145   // function. The returned slice always has the same number of elements
      // as the source slice.
      func fmap(fn func(Type) Type1, slice []Type) []Type1

      // The fold built-in functions fold over a slice of elements with the
```

```
150   // given function. It takes init, the second argument, and the first /
      // last item of the list and applies the function, then it takes the
      // penultimate item from the end and the result, and so on.
      // foldr and foldl differ in their evaluation order - foldr starts
      // at the last, foldl at the first element of the slice.
155   func foldr(fn func(Type, Type1) Type1, acc Type1, slice []Type) Type1
      func foldl(fn func(Type1, Type) Type1, acc Type1, slice []Type) Type1


      // The filter built-in function filters a slice with the given
      // function. If the function returns true on an element, the
160   // element will be added to the returned slice.
      func filter(fn func(Type) bool, slice []Type) []Type


      // ...
```

Source Code 4.1: Godoc for the new built-in functions

4.1.3 Public packages

To enable tooling support for the new built-in functions, they have to be registered in the 'go/*' packages. The only package that is affected by new built-ins is 'go/types'.

> Note that the 'go/*' family of packages, such as 'go/parser' and 'go/types', have no relation to the compiler. Since the compiler was initially written in C, the 'go/*' packages were developed to enable writing tools working with Go code, such as 'gofmt' and 'vet'.[41]

In the 'types' package, the built-ins have to be registered as such and as 'predeclared' functions:

```
107   // ...
      // A builtinId is the id of a builtin function.
      type builtinId int

110
      const (
        // universe scope
        _Append builtinId = iota
        _Prepend
115     _Fmap
```

```
    _Foldr
    _Foldl
    _Filter
    // ...
```

```
145  // ...
     var predeclaredFuncs = [...]struct {
       name     string
       nargs    int
       variadic bool
150    kind     exprKind
     }{
       _Append:  {"append", 1, true, expression},
       _Prepend: {"prepend", 2, false, expression},
       _Fmap:    {"fmap", 2, false, expression},
155    _Foldr:   {"foldr", 3, false, expression},
       _Foldl:   {"foldl", 3, false, expression},
       _Filter:  {"filter", 2, false, expression},
       _Cap:     {"cap", 1, false, expression},
       // ...
```

Source Code 4.2: Registering new built-in functions

This registration defines the type of the built-in — they are all expressions, as they return a value — and the number of arguments. After that, the type-checking and its associated tests have been implemented, but are not shown here. The implementation can be located in the 'src/go/types' package in the files 'builtins.go', 'builtins_test.go' and 'universe.go' See the git diff[70] to view the changes that have been made.

This concludes the type-checking for external tools. 'gopls' can be compiled against these changed public packages[3] and will then return errors if the wrong types are used. For example, when trying to prepend an integer to a string slice:

```
package main

import "fmt"
```

---
[3]See Appendix 11 for instructions on how to build Gopls.

```go
func main() {
  fmt.Println(prepend(3, []string{"hello", "world"}))
}
```

Gopls will report a type-checking error:

```
$ gopls check main.go
/tmp/playground/main.go:6:22-23: cannot convert 3 (untyped int constant)
↪  to string
```

4.1.4 Private packages

In the private packages - the actual compiler - the expressions have to be type-checked, ordered and transformed.

The type-checking process is similar to the one executed for external tools. Furthermore, during the type-checking process, the built-in function's return types are set and node types may be converted, if possible and necessary. An operation may expect it's arguments to be in `node.Left` and `node.Right`, which means type-checking will also need to move the argument nodes from their default location in `node.List` to `node.Left` and `node.Right`.

Ordering ensures the evaluation order and re-orders expressions. All of the new built-in functions will be evaluated left-to-right and there are no special cases to handle.

Transforming means changing the AST nodes from the built-in operation to nodes that the compiler knows how to translate to SSA. The actual algorithm that these functions use cannot be implemented in normal Go code, they have to be translated directly to AST nodes and statements.

There are more steps to compiling Go code, for example escape-checking, SSA conversion and a lot of optimisations. These are not necessary to implement and do not have a direct relation to the new built-ins.

The algorithms and part of the implementations for the built-in functions are covered in the following chapters[4].

---

[4]To see the full implementation, the git diff can be viewed[70].

fmap

To make the implementation in the AST easier, the algorithm will first be developed in Go, and then translated. Implementing fmap in Go is relatively simple:

```go
func fmap(fn func(Type) Type1, src []Type) (dest []Type1) {
  for _, elem := range src {
    dest = append(dest, fn(elem))
  }
  return dest
}
```

Source Code 4.3: fmap implementation in Go

However, there is room for improvement within that function. Instead of calling `append` at every iteration of the loop, the slice can be allocated with `make` at the beginning of the function. Thus, calls to grow the slice at runtime can be saved.

```go
func fmap(fn func(Type) Type1, src []Type) []Type1 {
  dest := make([]Type1, len(src))
  for i, elem := range src {
    dest[i] = fn(elem)
  }
  return dest
}
```

Source Code 4.4: Improved implementation of fmap

This algorithm can be translated to the following AST node:

```
3043    // ...
        // walkfmap rewrites the builtin fmap(f(in) out, []slice) to
3045    //
        //   init {
        //     dst = make([]out, len(slice))
        //     for i, e := range slice {
        //       dst[i] = f(e)
```

```
3050  //      }
      //    }
      //    dst
      //
      func walkfmap(n *Node, init *Nodes) *Node {
3055    // ...
```

Source Code 4.5: fmap AST translation[71]

The full AST code is not displayed here as, although the algorithm is simple, the AST translation is not as concise and more than 10 times the size. A demonstration on how a translation looks like will be introduced in Section 4.1.4. The full implementation of this function is referenced in the code block's caption.

prepend

The general algorithm for 'prepend' is:

```
func prepend(elem Type, slice []Type) []Type {
  dest := make([]Type, 1, len(src)+1)
  dest[0] = elem
  return append(dest, slice...)
}
```

Source Code 4.6: prepend implementation in Go

The call to `make(...)` creates a slice with the length of 1 and the capacity to hold all elements of the source slice, plus one. By allocating the slice with the full length, another slice allocation within the call to `append(...)` is saved. The element to prepend is added as the first element of the slice, and append will then copy the 'src' slice into 'dest'.

The implementation within 'walkprepend' reflects these lines of Go code, but as AST nodes:

```
3000  // ...
      // walkprepend rewrites the builtin prepend(elem, slice) to
      //
```

```
3005

3010
```

```
//   init {
//     dest := make([]<T>, 1, len(slice)+1)
//     dest[0] = elem
//     append(dest, slice...)
//   }
//   dest
//
func walkprepend(n *Node, init *Nodes) *Node {
  // ...
```

Source Code 4.7: prepend AST translation[72]

foldr and foldl

As outlined in Chapter 3.1.4, there will be two fold functions; foldr and foldl. foldr behaves exactly like its Haskell counterpart, while foldl behaves like foldl' in Haskell.

While the fold algorithms are most obvious when using recursion, due to performance considerations, an imperative implementation has been chosen:

```go
func foldr(fn func(Type, Type1) Type1, acc Type1, slice []Type) Type1 {
  for i := len(s) - 1; i >= 0; i-- {
    acc = fn(s[i], acc)
  }
  return acc
}

func foldl(fn func(Type1, Type) Type1, acc Type1, slice []Type) Type1 {
  for i := 0; i < len(s); i++ {
    acc = f(acc, s[i])
  }
  return acc
}
```

Source Code 4.8: fold implementation in Go

The code further clarifies the differences between the two different folds; the slice is

processed in reverse order for foldr (as it would be if this algorithm would have been implemented with recursion), and the order of arguments to the fold function is switched.

The AST walk translates fold to:

```
3108   // ...
       // walkfold rewrites the builtin fold function.
3110   // For the right fold:
       //   foldr(f(T1, T2) T2, a T2, s []T1) T2
       //
       //     init {
       //       acc = a
3115   //       for i := len(s) - 1; i >= 0; i-- {
       //         acc = f(s[i], acc)
       //       }
       //     }
       //     acc
3120   //
       // And the left fold:
       //   foldl(f(T2, T1) T2, a T2, s []T1) T2
       //
       //     init {
3125   //       acc = a
       //       for i := 0; i < len(s); i++ {
       //         acc = f(acc, s[i])
       //       }
       //     }
3130   //     acc
       func walkfold(n *Node, init *Nodes, isRight bool) *Node {
         // ...
```

Source Code 4.9: fold AST translation[73]

filter

Being a slice-manipulating function, filter also needs to traverse the whole slice in a for-loop. However, compared to the other newly built-in functions, the size for the target slice is unknown until all items have been traversed, which is why filter does not allow for the same optimisations as the other functions.

```
func filter(f func(Type) bool, s []Type) []Type {
  var dst []Type
  for i := range s {
      if f(s) {
        dst = append(dst, s[i])
      }
  }
}
```

Source Code 4.10: filter implementation in Go

And the same algorithm, but translated to AST statements:

```
3181  // ...
      // walkfilter rewrites the builtin filter function.
      //   filter(f(T) bool, slice []T) []T
      //
3185  //   init {
      //     dst = make([]out, 0)
      //     for i, e := range slice {
      //       if f(slice[i]) {
      //         dst = append(dst, slice[i])
3190  //       }
      //     }
      //   }
      //   dst
      //
3195  func walkfilter(n *Node, init *Nodes) *Node {
        // ...
```

Source Code 4.11: filter AST translation[74]

Writing the AST traversal

The previous chapters have all shown the function headers of the 'walk' functions that are used to traverse and rewrite the new built-ins. To illustrate how the actual implementation

of such an algorithm looks like in these functions, we provide a small example here. The full implementation of these algorithms can be viewed at the git diff[70].

This demonstration shows the translation of the statement

```
filtered := make([]T, 0)
```

into AST nodes, or rather the construction of these AST nodes. The type is simply a placeholder, as the AST construction uses the source slice's type. This source slice is another AST node of which the type can be obtained from.

```
// create the AST node for the first argument that is
// being passed to `make', the type:
makeType := nod(OTYPE, nil, nil)
makeType.Type = source.Type // use the type of the slice

// create the make(...) AST node
makeDest := nod(OMAKE, nil, nil)
// add the arguments (the type and an int constant 0
makeDest.List.Append(makeType, nodintconst(0)) // make([]<T>, 0))

// create the "variable" where the result of make will be stored
filtered := temp(source.Type)
// the final AST node that contains the statement
//   filtered = make([]<T>, 0)
final := nod(OAS, filtered, makeDest))
```

Source Code 4.12: Illustrating the difference between Go code and it's AST code

4.2 Functional Check

As discussed in Chapter 3.2, to assist writing purely functional code, a linter needs to be written that detects reassignments within a Go program.

To get a grasp about the issues this linter should report, the first step is to capture some examples, cases that should be matched against.

4.2.1 Examples

The simplest cases are standalone reassignments and assignment operators:

```
x := 5
x = 6 // forbidden
// or
var y = 5
y = 6   // forbidden
y += 6  // forbidden
y <<= 2 // forbidden
y++     // forbidden
```

Where the statements with a `// forbidden` comment should be reported.

Adding block scoping to this, shadowing the old variable needs to be allowed:

```
x := 5
{
  x = 6  // forbidden, changing the old value
  x := 6 // allowed, as this shadows the old variable
}
```

What should be illegal is to declare the variable first and then assign a value to it:

```
var x int
x = 6 // forbidden
```

The exception here are functions, as they need to be declared first in order to recursively call them:

```
var f func()
f = func() {
  f()
}
```

Furthermore, the linter also needs to be able to handle multiple variables at once:

```go
var f func()
x, f, y := 1, func() { f() }, 2
```

Loops should be reported too, as they are using reassignments internally:

```go
for i := 0; i < 5; i++ { // forbidden
  for i != 3 { // forbidden
    for { // allowed
      // ...
    }
  }
}
```

All the aforementioned examples and more can be found in the testcases for funcheck[75].

4.2.2 Building a linter

The Go ecosystem already provides an official library for building code analysis tools, the 'analysis' package from the Go Tools repository[76]. With this package, implementing a static code analyser is being reduced to writing the actual AST node analysis.

To define an analysis, a variable of type `*analysis.Analyzer` has to be declared:

```go
var Analyzer = &analysis.Analyzer{
  Name: "assigncheck",
  Doc:  "reports reassignments",
  Run:  func(*analysis.Pass) (interface{}, error)
}
```

The necessary steps are now adding the 'Run' function and registering the analyser in the `main()` function.

The 'Run' function takes an `*analysis.Pass` type. The Pass provides information about the package that is being analysed and some helper-functions to report diagnostics.

With 'analysis.Pass.Files' and the help of the 'go/ast' package, traversing the syntax tree of every file in a package is made extremely convenient:

```go
for _, file := range pass.Files {
  ast.Inspect(file, func(n ast.Node) bool {
    // node analysis here
  })
}
```

To implement funcheck as described, five different AST node types need to be taken care of. The simpler ones are `*ast.IncDecStmt`, `*ast.ForStmt` and `*ast.RangeStmt`. An 'IncDecStmt' node is a `x++` or `x--` expression and should always be reported. 'ForStmt' and 'RangeStmt' are similar; a 'RangeStmt' is a 'for' loop with the `range` keyword instead of an init-, condition- and post-stametent.

Both of these loop-types need to be reported explicitly as they do not show up as reassignments in the AST. Thus, the basic building block for the AST traversal is the following `switch` statement:

```go
31      // ...
        switch as := n.(type) {
        case *ast.ForStmt:
          // exception for `for { ... }`,
35        if as.Cond != nil || as.Init != nil || as.Post != nil {
            pass.Reportf(as.Pos(), "internal reassignment (for loop) in
            ↪  %q", renderFor(pass.Fset, as))
          }
          return true
        case *ast.RangeStmt:
40        pass.Reportf(as.Pos(), "internal reassignment (for loop) in
          ↪  %q", renderRange(pass.Fset, as))
          return true

        case *ast.IncDecStmt:
          pass.Reportf(as.Pos(), "inline reassignment of %s",
          ↪  render(pass.Fset, as.X))
45      // ...
```

Source Code 4.13: Handling the basic AST types in funcheck

The remaining two node types are `*ast.DeclStmt` and `*ast.AssignStmt`. They are not as simple to handle, which is why they are covered in their own chapters.

4.2.3 Detecting reassignments

To recapitulate, the goal of this step is to detect all assignments except blank identifiers (discarded values cannot be mutated) and function literals, if the function is declared in the last statement[5].

To detect such reassignments, funcheck iterates over all identifiers on the left-hand side of an assignment statement.

On the left-hand side of an assignment is a list of expressions. These expressions can be identifiers, index expressions (`*ast.IndexExpr`, for map and slice access), a 'star expression' (`*ast.StarExpr`[6]) or others.

If the expression is not an identifier, the assignment must be a reassignment, as all non-identifier expressions contain an already declared identifier. For example, the slice index expression `s[5]` is of type `*ast.IndexExpr`:

```go
// An IndexExpr node represents an expression followed by an index.
IndexExpr struct {
    X      Expr      // expression
    Lbrack token.Pos // position of "["
    Index  Expr      // index expression
    Rbrack token.Pos // position of "]"
}
```

Where `IndexExpr.X` is our identifier 's' (of type `*ast.Ident`) and a `IndexExpr.Index` is `5` (of type `*ast.BasicLit`).

As these nested identifiers already need to be declared beforehand (else they could not be used in the expression), all expressions on the left-hand side of an assignment that are not identifiers are reassignments.

Identifiers are the only expressions that can occur in declarations and reassignments. A naive approach would be to check for the colon in a short variable declaration (`:=`). However, as touched upon in Chapter 3.2.5, even short variable declarations may contain redeclarations, if at least one variable is new.

Thus, another approach is needed.

---

[5]This rule is to simplify the logic of the checker and make it easier for developers to read the code. It means that no code may be between `var f func` and `f = func() { ... }`.

[6]star expressions are expressions that are prefixed by an asterisk, dereferencing a pointer. For example `*x = 5`, if x is of type `*int`.

Every identifier (an AST node with type *ast.Ident) contains an object[7] that links to the declaration.

This declaration, of whatever type it may be, always has a position (and a corresponding function to retrieve that position) in the source file.

A reassignment is detected if an identifier's declaration position does not match the assignment's position (indicating that the variable is being assigned at a different place to where it is declared).

This is illustrated in the code block 4.14. What can be clearly seen is that in the assignment `y = 3`, y's declaration refers to the position of the first assignment `x, y := 1, 2`, the position where `y` has been declared.

```
Assignment "x, y := 1, 2": 2958101
        Ident "x": 2958101
                Decl "x, y := 1, 2": 2958101
        Ident "y": 2958104
                Decl "x, y := 1, 2": 2958101
Assignment "y = 3": 2958115
        Ident "y": 2958115
                Decl "x, y := 1, 2": 2958101
```

Source Code 4.14: Illustration of an assignment node and corresponding positions[78]

As this technique works on an identifier level, multi-variable declarations or assignments can be verified without any additional effort.

If a variable in a short variable declaration is being reassigned, the variable's 'Declaration' field will point to the original position of its declaration, which can be easily detected (as shown in code block 4.14).

4.2.4 Handling function declarations

In contrast to all other variable types, function variables may be 'reassigned' once. As discussed in Chapter 3.2.5, this is to allow recursive function literals. Detecting and not reporting these assignments is a two-step process, as two consecutive AST nodes need to be inspected.

---

[7]'An object describes a named language entity such as a package, constant, type, variable, function (incl. methods), or a label'[77].

The first step is to detect function declarations; statements of the form `var f func()`. Should such a statement be encountered, its position is saved for the following AST node.

In the consecutive AST node it is ensured that, if the node is an assignment and the assignee identifier is of type function literal, the position matches the previously saved one.

The position of the declaration and AST node structure can be seen in 4.15

```
Declaration "var f func() int": 2958142
        Ident "f func() int": 2958146
Assignment "f = func() int { return y }": 2958160
        Ident "f": 2958160
                Decl "f func() int": 2958146
```

Source Code 4.15: Illustration of a function literal assignment[78]

With this technique it is possible to exempt functions from the reassignment rule.

4.2.5 Testing Funcheck

The analysis-package is distributed with a sub-package 'analysistest'. This package makes it extremely simple to test a code analysis tool.

By providing test data and the expected messages from funcheck in a structured way, all that is needed to test funcheck is:

```
package assigncheck

import (
        "testing"

        "golang.org/x/tools/go/analysis/analysistest"
)

func TestRun(t *testing.T) {
        analysistest.Run(t, analysistest.TestData(), Analyzer)
}
```

Source Code 4.16: Testing a code analyser with the 'analysistest' package

The library expects the test data in the current directory in a folder named 'testdata' and then spawns and executes the analyser on the files in that folder. Comments in those files are used to describe the expected message:

```
x := 5
fmt.Println(x)
x = 6 // want `^reassignment of x$`
fmt.Println(x)
```

This will ensure that on the line `x = 6` an error message is reported that says 'reassignment of x'.

5 Application

5.1 Demonstration of the new built-in functions

The new built-in functions can be used in the same manner as the existing built-ins like
`append`. This example demonstrates this by modifying a slice of integers, although they
can be used to modify a slice of any type.

```go
package main

import (
  "fmt"
  "strconv"
)

// Counterpart to Haskell's `derive Show` through code generation
//go:generate stringer -type parity

type parity int

const even parity = 0
const odd parity = 1

// shouldBe returns a function that returns true if an int is of the
// given parity
func shouldBe(p parity) func(i int) bool {
  return func(i int) bool {
    return i%2 == int(p)
  }
}

func main() {
  lst := []int{1, 2, 3, 4, 5}
  lstMult := fmap(func(i int) int { return i * 5 }, prepend(0, lst))
```

```
addToString := func(s string, i int) string {
  return s + strconv.Itoa(i) + " "
}
// fold over even / odd numbers and add them to a string
evens := foldl(addToString, even.String()+": ",
  filter(shouldBe(even), lstMult))
odds := foldl(addToString, odd.String()+": ",
  filter(shouldBe(odd), lstMult))

fmt.Println(evens, odds) // even: 0 10 20  odd: 5 15 25
}
```

Source Code 5.1: Demonstration of the new built-in functions

5.2 Refactoring the Prettyprint Package

The code blocks 4.14 and 4.15 have been generated by a small package 'prettyprint' contained in the funcheck repository.

To see how the newly built-in functions and funcheck can be used, this 'prettyprint' package can be refactored to a purely functional version. The current version of the package is written in what could be considered idiomatic Go[1].

The prettyprinter is based on the same framework as assigncheck[2], but instead of reporting anything, it prints AST information to stdout.

Similarly to assigncheck, the main logic of the package is within a function literal that is being passed to the `ast.Inspect` function.

Prettyprint only checks two AST node types, ∗`ast.DeclStmt` and ∗`ast.AssignStmt` (declarations and assignments).

For example, for the program

---

[1]There is no exact definition of what idiomatic Go is, so this interpretation could be challenged. It is idiomatic Go code to the author.

[2]Assigncheck is the main package for funcheck and checks the reassignments

```
package main

import "fmt"

func main() {
  x, y := 1, 2
  y = 3
  fmt.Println(x, y)
}
```

the following AST information is printed:

```
Assignment "x, y := 1, 2": 2958101
     Ident "x": 2958101
          Decl "x, y := 1, 2": 2958101
     Ident "y": 2958104
          Decl "x, y := 1, 2": 2958101
Assignment "y = 3": 2958115
     Ident "y": 2958115
          Decl "x, y := 1, 2": 2958101
```

To refactor it to a purely functional version, funcheck can be used to list reassignments:

```
$> funcheck .
prettyprint.go:20:2: internal reassignment (for loop) in
↪    "for _, file := range pass.Files { ... }"
prettyprint.go:42:2: internal reassignment (for loop) in
↪    "for i := range decl.Specs { ... }"
prettyprint.go:67:2: internal reassignment (for loop) in
↪    "for _, expr := range as.Lhs { ... }"
```

As can be seen in the output, the package uses 3 `for` loops to range over slices. However, there are no other reassignments of variables in the code.

The code to print declarations, which causes the second lint message, is as shown in code block 5.2.

```go
func checkDecl(as *ast.DeclStmt, fset *token.FileSet) {
  fmt.Printf("Declaration %q: %v\n", render(fset, as), as.Pos())
  decl, ok := as.Decl.(*ast.GenDecl)
  if !ok {
    return
  }

  for i := range decl.Specs {
    val, ok := decl.Specs[i].(*ast.ValueSpec)
    if !ok {
      continue
    }

    if val.Values != nil {
      continue
    }

    if _, ok := val.Type.(*ast.FuncType); !ok {
      continue
    }

    fmt.Printf("\tIdent %q: %v\n", render(fset, val),
    ↪  val.Names[0].Pos())
  }
}
```

Source Code 5.2: Pretty-printing declarations in idiomatic Go

To convert this for-loop appropriately, the new built-in 'foldl' can be used. To recapitulate, the 'foldl' function is being defined as:

```go
func foldl(fn func(Type1, Type) Type1, acc Type1, slice []Type) Type1
```

As 'foldl' requires a return type, a dummy type 'null" can be introduced, which is just an empty struct:

```go
type null struct{}
```

Now the code within the for loop can be used to create a function literal:

```go
check := func(_ null, spec ast.Spec) (n null) {
  // implementation
}
```

There are two subtleties in regards to the introduced null type: First, the null value that is being passed as an argument is being discarded by the use of an empty identifier. Secondly, the return value is 'named', which means the variable 'n' is already declared in the function block. Because of this, 'naked returns' can be used, so there is no need to specify which variable is being returned.

The code snippet 5.2 can be translated to

```go
func checkDecl(as *ast.DeclStmt, fset *token.FileSet) {
  fmt.Printf("Declaration %q: %v\n", render(fset, as), as.Pos())

  check := func(_ null, spec ast.Spec) (n null) {
    val, ok := spec.(*ast.ValueSpec)
    if !ok {
      return
    }

    if val.Values != nil {
      return
    }

    if _, ok := val.Type.(*ast.FuncType); !ok {
      return
    }

    fmt.Printf("\tIdent %q: %v\n", render(fset, val),
    ↪ val.Names[0].Pos())
    return
  }

  if decl, ok := as.Decl.(*ast.GenDecl); ok {
    _ = foldl(check, null{}, decl.Specs)
  }
}
```

Source Code 5.3: Pretty-printing declarations in functional Go

The for-loop has been replaced by a 'foldl', where a function closure that contains the actual processing is passed.

While this still looks similar to the original example, this is mostly due to the 'if' statements. In Haskell, pattern matching would be used and nil checks could be omitted entirely. Also, as Haskell's type system is more advanced, the handling of those types would be different too.

However, the goal of this thesis is to make functional code look more familiar to programmers that are used to imperative code. And while it may not look like it, the code does not use any mutation of variables[3], for loops or global state. Therefore, it can be concluded that this snippet is purely functional as per the definition from Chapter 3.2.1.

5.3 Quicksort

In Chapter 1.1, a naive implementation of the Quicksort sorting algorithm has been introduced. Implementing this algorithm in Go is now straightforward and the similarities between the Haskell implementation and the functional Go implementation are striking:

---

[3]Libraries may do, but the scope is not to rewrite any existing libraries.

```go
func quicksort(p []int) []int {
  if len(p) == 0 {
    return []int{}
  }

  lesser := filter(func(x int) bool { return p[0] > x }, p[1:])
  greater := filter(func(x int) bool { return p[0] <= x }, p[1:])

  return append(quicksort(lesser), prepend(p[0], quicksort(greater))...)
}
```

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort []     = []
quicksort (p:xs) = (quicksort lesser) ++ [p] ++ (quicksort greater)
    where
        lesser  = filter (< p) xs
        greater = filter (>= p) xs
```

Source Code 5.4: Quicksort implementations compared

Again, the Go implementation bridges the gap between being imperative and functional, while still being obvious about the algorithm. Furthermore, as expected, when inspecting the code with funcheck, no non-functional constructs are reported.

5.4 Comparison to Java Streams

In Java 8, concepts from functional programming have been introduced to the language. The major new feature was Lambda Expressions — anonymous function literals — and streams. Streams are an abstract layer to process data in a functional way, with 'map', 'filter', 'reduce' and more.

Java Streams are similar to the new built-in functions in this thesis:

```java
List<Integer> even = list.stream()
  .filter(x -> x % 2 == 0)
  .collect(Collectors.toList());
```

```go
even := filter(
  func(x int) bool { return x%2 == 0 },
  list)
```

Source Code 5.5: Comparison Java Streams and functional Go

The lambda-syntax in Java is more concise than Go's function literals, where the complete function header has to be provided[4].

However, the conversion to a stream and back to a list (with `list.stream()` and `.collect(Collectors.toList())`) is not required in Go, reducing the mental overhead for the programmer.

Apart from syntactical differences, Java Streams contain all the functions that have been added as built-ins to Go too, and a lot more.

On the other hand, Java's Syntax is arguably more complex than Go. An indicator for this might be the language specification; Go's Language Specification is roughly 110 pages, while Java's specification spans more than 700 pages[5], more than 6 times the size.

The consideration of which language to choose comes down to the experience with either language. An experienced Java programmer will find it easier to start with Java's toolset, while programmers coming from a C background may choose Go over Java.

---

[4]There is an open proposal to add a lightweight anonymous function syntax to Go 2, which, if implemented, would resolve this verbosity[79]

[5]The Java 8 Specification is 724[80], the Java 14 Specification 774[81] pages.

6 Results

To learn functional programming without being introduced to a new syntax at the same time ensures that programmers can fully concentrate on functional concepts. Although Go already supported a functional programming style, the programmer may not have known if the code was purely functional or if there still were imperative constructs embedded.

In the last chapters, functional purity has been defined as a law based on two rules; immutability and function purity. Immutability means that once assigned, a variable's value never changes. Function purity entails that functions do not have side effects and their return value is solely influenced by the function's parameters.

It has been shown that although purely functional languages like Haskell aim to be completely pure, this objective is difficult to accomplish. The reason for this are Input / Output actions; user input, network connections, randomness and time are all impure. Haskell wraps these impure functions in the IO monad, which is a way to work around the compiler's optimisations based on functional purity. In addition, although the IO monad does not make impure functions pure, it does serve as documentation to its users ('if the function has IO, it is impure') and guarantees a certain execution order.

Go on the other hand does not have this issue, as the Go compiler does not optimise execution based on purity guarantees. Having a similar construct like the IO monad in Go would as such only serve documentation purposes. Because of this, the decision has been taken to ignore the impurity that is implied with IO actions.

Apart from IO, to achieve functional purity, the global state of a program should not influence the return values of specific function. This ties into immutablitiy; if global state can not be mutated, it can also not influence or change the result value of a function.

Based on these observations, a static code analysis tool has been developed that reports all reassignments of variables. In other words, it forbids the usage of the regular assignment operator (=), only allowing the short variable declarations (:=). However, the experienced Go developer may know that the := operator can also reassign previously declared variables, implying that the solution to the problem is not as simple as forbidding the assignment operator. Further, there are many more edge cases that have been detected with careful testing: To recursively call function literals, they must be declared beforehand (before assigning the actual function to it) because of Go's scoping rules. Additionally,

exceptions had to be made for the blank identifier (`_`) and variables that are declared outside of the current file.

With all of this in place, an algorithm has been chosen that is based on the identifier's declaration position. In the AST that is being checked, every identifier node has a field which contains the position of it's declaration. If this does not match the current identifier's position, the operation must be a reassignment. The resulting binary, called 'funcheck', successfully reports such reassignments:

```go
s := "hello world"
fmt.Println(s)
s = "and goodbye"
fmt.Println(s)
```

```
$> funcheck .
file.go:3:2: reassignment of s
```

This linter can be used and executed against any Go package. To eliminate the reported errors, code has to be rewritten in what ends up being purely functional code.

However, functional code often relies heavily on lists and list-processing functions. Although Go does not have a built-in list datatype, Go's slices, an abstraction built on top of arrays, mitigate a lot of downsides when comparing regular arrays to lists[1].

What Go's slices lack on the other hand are the typical higher-order functions like 'map', 'filter' or 'reduce'. These are commonly used in functional programming and most languages contain implementations of these functions already — Go does not.

Due to the lack of polymorphism, writing implementations for these functions would result in a lot of duplicated code. To mitigate this issue, the most common higher-order functions have been added to the list of Go's built-in functions, which are registered, type-checked and implemented within the Go compiler. As these are handled directly at compile time, built-in functions may be polymorphic, for example allowing the programmer to use the same 'filter' function for all list-types.

To determine which higher-order functions are most commonly used, the most popular open-source Haskell projects (pandoc, shellcheck and purescript, to name a few) have been analysed. As a result, 'fmap', 'fold', 'filter' and 'prepend' ('cons') have been added as built-ins into the compiler. These functions make it easier to write purely functional

---

[1] Arrays / Slices and Lists have a different runtime behaviour (indexing, adding or removing elements). However, the performance of the code was not considered to be in scope for this thesis.

code in Go, in turn helping the programmer to learn functional programming with a familiar language and syntax.

While implementing these functions in a regular Go program would be a matter of minutes, adding them to the Go compiler is not as simple. To illustrate, the functions have been written out in regular Go in the chapters 4.1.4 to 4.1.4 and are 33 lines of code, all functions combined. In the Go compiler, it is necessary to register the functions, type-check the calls and manipulate the AST instead of writing the algorithm in Go code directly. This took more than 800 lines of code to do so.

As a result, using these functions is equal to using any other built-in function: there is documentation in Godoc, type-checking support in the language server[2] and in the compilation phase, as well as a polymorphic function header, allowing the programmer to call the function with any allowed type.

Demonstrations of these functions and how functional Go code looks like can be seen in Chapter 5.

With these additions to Go and its ecosystem, aspiring functional programmers can fully concentrate of the concepts of functional programming while keeping a familiar syntax at hand. However, it should not be considered a fully featured purely functional programming language. Rather, it should serve as a starting point and make the transition to a language like Haskell easier.

Differences in the syntax between Haskell and Go exemplify why purely functional programming languages have a distinct syntax compared to imperative or object-oriented languages. Many constructs can be expressed more concisely in Haskell, without the additional overhead that many programming languages, including Go, introduce.

Using 'the right tool for the job' is important, and this paper shows that imperative or object-oriented programming languages are not the right tool for production-grade functional programming. However, they can serve as a good starting point and help transitioning to it.

---

[2]If the language server (gopls) is compiled against the modified version of Go, as described in Appendix 11

7 Discussion

The aforementioned extensions to the Go language and its tooling should be a help to learn functional programming. I believe that through these extensions it is easier to write purely functional code in Go, enabling a developer to learn functional programming with a familiar syntax in an obvious way. Here, Go's simplicity and verbosity are a key differentiator to other languages. Instead of having as many features as possible to support every usecase, Go has been designed with simplicity in mind[1].

In many cases, this leads to more 'verbose' code — more lines of code compared to a similar implementation in other languages. However, I argue that, especially for the first steps in functional programming,

> *Clear is better than clever[82]*

Staying in touch with this core Go principle, this results in functional code that may be verbose, but easy to read and understand.

It should be clear that the result is not a 'production-ready' functional programming language. It is a language to help getting started with functional programming; either by re-implemeting pieces of code that have not been clear in how they work, or by taking an imperative block of code and refactoring it to make it purely functional.

In many cases, the resulting code will still look familiar to the imperative counterpart, even if 'funcheck' assures that it is purely functional. This, I believe, bridges the gap that developers usually have to overcome by themselves.

To be a purely functional programming language, Go is missing too many features that would be required to write concise functional code. The very basic type system[2], no advanced pattern matching and only explicit currying are all examples why Go is not useful in day-to-day functional programming.

At the same time, the obvious nature of Go is exactly because it is missing all of these features. The Go team explicitly tries not to include too many features within the language in order to keep the complexity of code to a minimum[83]. The simplicity of the

---

[1]For example, Go only has 25 keywords, compared to 37 in C99 and 84 in C++11

[2]Not only does Go not have polymorphism (yet), Go's type system is simple by design: there are no implicit type conversions, no sum types (tagged unions, variant) and almost no type inference.

language is a key feature of Go and an important reason why it was chosen to implement the ideas in the first place. Especially for learning new concepts, hiding implementations and ideas behind features may not be what is desired and helpful.

On another note, what has not been an aspect in this thesis is performance. Go by itself is relatively performant, however functional constructs, for example recursive function calls, come with a performance cost. While in purely functional languages this can be optimised, Go cannot or does not want to do these optimisations[3].

While the newly built-in functions do not have any low-hanging fruits in regards to optimisation, they would benefit from more aggressive inlining[85], for example. Benchmarking and optimising these functions was out of scope for this thesis, but may be tackled in the future.

The number one issue that still exists is the simple type system. Not only the lack of polymorphism, which has been mitigated slightly by providing the most used higher-order functions as built-ins, but also the lack of algebraic data types, especially sum types.

Algebraic data types can be split up into two groups, product types and sum types. Most product types can be built with Go too; records are basically equal to structs, and tuples are not needed too often, as functions can just return multiple values. Sum types however are not available in Go at all. It is possible to imitate sum types in Go with interfaces (see the example code in 5.1), but the compiler does not ensure that all cases are matched against[4]. This may be an interesting area for further research and implementation possibilities.

---

[3]For example, with tail call optimisation, the Go team explicitly decided not to do it because the stack trace would be lost[84]

[4]There is tooling available to check this[86], but it is third party and not built into the language

Bibliography

[1] E. Dietrich and P. Smacchia. (Apr. 2020). The history of C# - C# guide, [Online]. Available: `https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-version-history#c-version-30` (visited on 06/05/2020).

[2] D. Robinson. (Sep. 2017). The incredible growth of pyhon, [Online]. Available: `https://stackoverflow.blog/2017/09/06/incredible-growth-python/?_ga=2.199625454.1908037254.1532442133-221121599.1532442133` (visited on 02/27/2020).

[3] 'Stack overflow developer survey 2019,' Stack Overflow, Tech. Rep., 2019. [Online]. Available: `https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted` (visited on 02/27/2020).

[4] S. Klabnik, C. Nichols, and R. Community. (2018). Functional language features, [Online]. Available: `https://doc.rust-lang.org/book/ch13-00-functional-features.html`.

[5] R. Quinlivan. (Nov. 2019). Why everyone should learn functional programming today, [Online]. Available: `https://medium.com/better-programming/why-everyone-should-learn-functional-programming-today-c96a5b10d27d` (visited on 05/22/2020).

[6] E. Normand. (Jul. 2019). Why functional programming, [Online]. Available: `https://purelyfunctional.tv/article/why-functional-programming/` (visited on 05/22/2020).

[7] A. Piirainen. (Jan. 2020). Why you should learn functional programming, [Online]. Available: `https://dev.to/anssip/why-you-should-learn-functional-programming-3h2g` (visited on 05/22/2020).

[8] A. MacGregor. (Jun. 2018). You should learn functional programming in 2018, [Online]. Available: `https://dev.to/allanmacgregor/you-should-learn-functional-programming-in-2018-4nff` (visited on 05/22/2020).

[9] T. Jelvis. (2014). Why should i learn a functional programming language, [Online]. Available: `https://www.quora.com/Why-should-I-learn-a-functional-programming-language` (visited on 05/22/2020).

[10] 'TIOBE Index for May 2020,' TIOBE, Tech. Rep., May 2020. [Online]. Available: `https://www.tiobe.com/tiobe-index/` (visited on 05/22/2020).

[11] Wikipedia contributors. (Feb. 2020). Comparison of functional programming languages, [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Comparison_of_functional_programming_languages&oldid=939648685` (visited on 03/04/2020).

[12] n/a. (Jun. 2017). Why is Haskell so hard to learn? [Online]. Available: `https://www.quora.com/Why-is-Haskell-so-hard-to-learn` (visited on 03/04/2020).

[13] I. Connolly. (Feb. 2014). A lazy evaluation, [Online]. Available: `http://connolly.io/posts/lazy-evaluation/` (visited on 03/04/2020).

[14] n/a. (Oct. 2018). Haskell's steep learning curve, [Online]. Available: `https://www.quora.com/How-much-of-Haskells-steep-learning-curve-is-related-to-the-number-of-terms-one-has-to-memorize?share=1` (visited on 03/04/2020).

[15] HaskellWiki contributors. (Sep. 2007). Humor/learningcurve — haskellwiki, [Online]. Available: `https://wiki.haskell.org/index.php?title=Humor/LearningCurve&oldid=15543` (visited on 03/04/2020).

[16] HaskellWiki contributors. (Feb. 2020). Introduction — haskellwiki, [Online]. Available: `https://wiki.haskell.org/index.php?title=Introduction&oldid=63206` (visited on 03/01/2020).

[17] J. Kincaid. (Nov. 2009). Google's Go: A New Programming Language That's Python Meets C++, [Online]. Available: `https://techcrunch.com/2009/11/10/google-go-language/` (visited on 02/29/2020).

[18] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `https://golang.org/doc/faq#Does_Go_have_a_ternary_form` (visited on 05/21/2020).

[19] The Go Authors. (Feb. 2020). http package - go.dev, [Online]. Available: `https://pkg.go.dev/net/http@go1.14?tab=doc#example-HandleFunc` (visited on 03/06/2020).

[20] D. Cheney. (Oct. 2014). Functional options for friendly APIs, [Online]. Available: `https://dave.cheney.net/2014/10/17/functional-options-for-friendly-apis` (visited on 03/06/2020).

[21] F. C. Flores. (Dec. 2015). Functional Go? Youtube, [Online]. Available: `https://www.youtube.com/watch?v=ouyHp2nJl0I` (visited on 03/06/2020).

[22] I. L. Taylor. (Nov. 2017). proposal: Go 2: add become statement to support tail calls, Github, [Online]. Available: `https://github.com/golang/go/issues/22624` (visited on 03/06/2020).

[23] A. Schlesinger. (Nov. 2019). go-functional/core, Github, [Online]. Available: `https://github.com/go-functional/core/tree/700b20aec09da808a67cc29ae2c54ad64f842851` (visited on 03/06/2020).

[24] A. Schlesinger. (Nov. 2019). go-functional/core, Github, [Online]. Available: `https://github.com/go-functional/core/blob/700b20aec09da808a67cc29ae2c54ad64f842851/README.md` (visited on 03/06/2020).

[25] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `https://golang.org/doc/faq#no_pointer_arithmetic` (visited on 06/05/2020).

[26] A. Gerrand. (Jan. 2011). Go Slices: usage and internals, [Online]. Available: `https://blog.golang.org/go-slices-usage-and-internals` (visited on 02/29/2020).

[27] The Go Authors. (Jan. 2020). SliceTricks, [Online]. Available: `https://github.com/golang/go/wiki/SliceTricks/a87dff31b0d8ae8f26902b0d2dbe7e4e77c7e6e3` (visited on 03/04/2020).

[28] T. Jelvis. (Sep. 2014). Why are lists so heavily used in most, if not all, functional programming languages? [Online]. Available: `https://www.quora.com/Why-are-lists-so-heavily-used-in-most-if-not-all-functional-programming-languages?share=1` (visited on 05/22/2020).

[29] HaskellWiki contributors. (Nov. 2019). How to work on lists — haskellwiki, [Online]. Available: `https://wiki.haskell.org/index.php?title=How_to_work_on_lists&oldid=63130` (visited on 03/04/2020).

[30] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec#Boolean_types` (visited on 03/02/2020).

[31] R. Pike. (Nov. 2015). Go Proverbs - Rob Pike - Gopherfest - November 18, 2015, Youtube, [Online]. Available: `https://www.youtube.com/watch?v=PAAkCSZUG1c&t=7m36s` (visited on 04/10/2020).

[32] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `https://golang.org/doc/faq#convert_slice_with_same_underlying_type` (visited on 05/26/2020).

[33] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `https://golang.org/doc/faq#convert_slice_of_interface` (visited on 05/26/2020).

[34] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec#Built-in_functions` (visited on 04/10/2020).

[35] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec` (visited on 03/02/2020).

[36] R. Pike. (Jun. 2012). Less is exponentially more, [Online]. Available: `https://comm andcenter.blogspot.com/2012/06/less-is-exponentially-more.html` (visited on 04/10/2020).

[37] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `htt ps://golang.org/doc/faq#creating_a_new_language` (visited on 04/10/2020).

[38] R. Pike. (Mar. 2014). Why does go programs compile faster than java or c#? [Online]. Available: `https://groups.google.com/d/msg/golang-nuts/al4iu FXLPeA/PYNcUQO_uAEJ` (visited on 04/10/2020).

[39] The Go Authors. (Nov. 2019). go/syntax.go at go1.14 - golang/go, [Online]. Available: `https://github.com/golang/go/blob/go1.14/src/cmd/compile/internal/gc /syntax.go#L18` (visited on 03/20/2020).

[40] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `https://golang.org/doc/faq#different_syntax` (visited on 04/10/2020).

[41] The Go Authors. (Jul. 2018). Introduction to the Go compiler, [Online]. Available: `https://github.com/golang/go/blob/go1.14/src/cmd/compile/README.md` (visited on 04/10/2020).

[42] n/a. (Mar. 2020). Search - stars:>1000 language:Haskell, Github, [Online]. Available: `https://github.com/search?l=&o=desc&q=stars%3A%3E1000+language%3AHask ell&s=stars&type=Repositories` (visited on 03/12/2020).

[43] V. Holen and contributors. (Mar. 2020). koalaman/shellcheck: ShellCheck, a static analysis tool for shell scripts, Github, [Online]. Available: `https://github.com/koa laman/shellcheck/tree/68a03e05e5e030d7274c712ffa39768310e70f8a` (visited on 03/12/2020).

[44] J. MacFarlane. (Mar. 2020). jgm/pandoc: Universal Markup Converter, Github, [Online]. Available: `https://github.com/jgm/pandoc/tree/91f2bcfe73fa3a489 654ee74bca02e24423dc5c0` (visited on 03/12/2020).

[45] The PostgREST contributors. (Mar. 2020). PostgREST/postgrest: REST API for any Postgres database, Github, [Online]. Available: `https://github.com/Post gREST/postgrest/tree/dea57bd1bec9ba5c4e438de80b2017eee4a30a40` (visited on 03/12/2020).

[46] Github and contributors. (Mar. 2020). github/semantic: Parsing, analyzing, and comparing source code across many languages, Github, [Online]. Available: `https: //github.com/github/semantic/tree/124b45d36d7f81e45a9aa3aef588fd5e132 fb6fd` (visited on 03/12/2020).

[47] The PureScript contributors. (Mar. 2020). purescript/purescript: A strongly-typed language that compiles to JavaScript, Github, [Online]. Available: `https://github .com/purescript/purescript/tree/183fc22549011804d973e01654e354b728f2b c70` (visited on 03/12/2020).

[48] The Elm Authors. (Mar. 2020). elm/compiler: Compiler for Elm, a functional language for reliable webapps, Github, [Online]. Available: `https://github.com /elm/compiler/tree/66c72f095e6da255bde8df6a913815a7dde25665` (visited on 03/12/2020).

[49] Github and contributors. (Mar. 2020). facebook/haxl: A Haskell library that simplifies access to remote data, such as databases or web-based services, Github, [Online]. Available: `https://github.com/facebook/Haxl/tree/0009512345fbd95fe1c745 414fffed6c63ccd1aa` (visited on 03/12/2020).

[50] HaskellWiki contributors. (Nov. 2019). Functor, [Online]. Available: `https://wiki .haskell.org/index.php?title=Functor&oldid=63127` (visited on 04/10/2020).

[51] M. Nic. (n/d). Haskell : map, [Online]. Available: `http://zvon.org/other/haskel l/Outputprelude/map_f.html` (visited on 04/10/2020).

[52] The Go Authors. (Feb. 2020). fmt package - go.dev, [Online]. Available: `https://p kg.go.dev/fmt@go1.14?tab=doc` (visited on 04/12/2020).

[53] n/a. (n/d). common-lisp - Sketching cons cells | common lisp Tutorial, [Online]. Available: `https://riptutorial.com/common-lisp/example/17740/sketching- cons-cells` (visited on 04/12/2020).

[54] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec#Identifiers` (visited on 04/12/2020).

[55] Wikipedia contributors. (Feb. 2020). Fold (higher-order function) - Wikipedia, [Online]. Available: `https:en.wikipedia.org/w/index.php?title=Fold_(highe r-order_function)&oldid=941001801` (visited on 04/12/2020).

[56] HaskellWiki contributors. (Mar. 2019). Foldr foldl foldl' — haskellwiki, [Online]. Available: `https://wiki.haskell.org/index.php?title=Foldr_Foldl_Foldl%2 7&oldid=62842` (visited on 04/07/2020).

[57] The golangci-lint contributors. (Mar. 2020). golangci/golangci-lint - Linters runner for Go, [Online]. Available: `https://github.com/golangci/golangci-lint/comm it/4958e50dfe8c95bebab5eaf360a3bc1fdc9574fe` (visited on 04/18/2020).

[58] A. Sabry, 'What is a purely functional language?' *Journal of Functional Programming*, vol. 8, no. 1, pp. 1–22, 1998. DOI: `10.1017/S0956796897002943`.

[59] Wikipedia contributors. (Aug. 2019). Purely functional programming — Wikipedia, the free encyclopedia, [Online]. Available: `https://en.wikipedia.org/w/inde x.php?title=Purely_functional_programming&oldid=910398359` (visited on 04/18/2020).

[60] Wikipedia contributors. (Feb. 2020). Final (java) — Wikipedia, the free encyclopedia, [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Final_(Ja va)&oldid=941659079` (visited on 04/25/2020).

[61] HaskellWiki contributors. (Mar. 2020). Io inside — haskellwiki, [Online]. Available: `https://wiki.haskell.org/index.php?title=IO_inside&oldid=63262` (visited on 04/26/2020).

[62] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec#Short_variable_declarations` (visited on 05/03/2020).

[63] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec#Operators_and_punctuation` (visited on 05/03/2020).

[64] The Go Authors. (Jan. 2020). The Go Programming Language Specification, [Online]. Available: `https://golang.org/ref/spec#Declarations_and_scope` (visited on 05/08/2020).

[65] A. Gerrand. (Mar. 2011). Godoc: documenting Go code, [Online]. Available: `https://blog.golang.org/godoc-documenting-go-code` (visited on 03/13/2020).

[66] The Go Authors. (Dec. 2019). gopls documentation, [Online]. Available: `https://github.com/golang/tools/blob/0b43622770f0bce9eb6c5d0539003ebbc68b9c70/gopls/README.md` (visited on 03/20/2020).

[67] R. Cox. (Jun. 2014). Go 1.4 "Internal"Packages, [Online]. Available: `https://docs.google.com/document/d/1e8kOo3r51b2BWtTs_1uADIA5djfXhPT36s6eHVRIvaU/edit` (visited on 04/01/2020).

[68] The Go Authors. (Mar. 2020). builtin package - go.dev, [Online]. Available: `https://pkg.go.dev/builtin@go1.14?tab=doc` (visited on 03/13/2020).

[69] The Go Authors. (Apr. 2019). go/builtin.go at go1.14 - golang/go, [Online]. Available: `https://github.com/golang/go/blob/go1.14/src/builtin/builtin.go` (visited on 03/13/2020).

[70] R. Rüttimann. (May 2020). Comparing go1.14...bachelor-thesis - tommyknows/go, [Online]. Available: `https://github.com/tommyknows/go/compare/go1.14...tommyknows:bachelor-thesis` (visited on 05/31/2020).

[71] R. Rüttimann. (May 2020). go/walk.go at bachelor-thesis - tommyknows/go, [Online]. Available: `https://github.com/tommyknows/go/blob/bachelor-thesis/src/cmd/compile/internal/gc/walk.go#L3055` (visited on 05/31/2020).

[72] R. Rüttimann. (May 2020). go/walk.go at bachelor-thesis - tommyknows/go, [Online]. Available: `https://github.com/tommyknows/go/blob/bachelor-thesis/src/cmd/compile/internal/gc/walk.go#L3011` (visited on 05/31/2020).

[73] R. Rüttimann. (May 2020). go/walk.go at bachelor-thesis - tommyknows/go, [Online]. Available: `https://github.com/tommyknows/go/blob/bachelor-thesis/src/cmd/compile/internal/gc/walk.go#L3132` (visited on 05/31/2020).

[74] R. Rüttimann. (May 2020). go/walk.go at bachelor-thesis - tommyknows/go, [Online]. Available: `https://github.com/tommyknows/go/blob/bachelor-thesis/src/cmd/compile/internal/gc/walk.go#L3196` (visited on 05/31/2020).

[75] R. Rüttimann. (May 2020). funcheck/example.go at master - tommyknows/funcheck, [Online]. Available: `https://github.com/tommyknows/funcheck/blob/master/assigncheck/testdata/example.go` (visited on 05/09/2020).

[76] The Go Authors. (May 2020). analysis package - go.dev, [Online]. Available: `https://pkg.go.dev/golang.org/x/tools@v0.0.0-20200509030707-2212a7e161a5/go/analysis?tab=doc` (visited on 05/09/2020).

[77] The Go Authors. (Jan. 2017). go/scope.go at go1.14 - golang/go, [Online]. Available: `https://github.com/golang/go/blob/go1.14/src/go/ast/scope.go#L64` (visited on 05/10/2020).

[78] R. Rüttimann. (May 2020). prettyprint package - go.dev, [Online]. Available: `https://github.com/tommyknows/funcheck/blob/master/prettyprint/README.md` (visited on 05/31/2020).

[79] D. Neil. (Aug. 2017). proposal: Go 2: Lightweight anonymous function syntax, Github, [Online]. Available: `https://github.com/golang/go/issues/21498` (visited on 05/16/2020).

[80] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. (Feb. 2015). The Java Language Specification - Java SE 8 Edition, [Online]. Available: `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf` (visited on 05/17/2020).

[81] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, and G. Bierman. (Feb. 2020). The Java Language Specification - Java SE 14 Edition, [Online]. Available: `https://docs.oracle.com/javase/specs/jls/se14/jls14.pdf` (visited on 05/17/2020).

[82] D. Cheney. (Jul. 2019). Clear is better than clever, [Online]. Available: `https://dave.cheney.net/2019/07/09/clear-is-better-than-clever` (visited on 05/21/2020).

[83] The Go Authors. (Sep. 2019). Frequently Asked Questions, [Online]. Available: `https://golang.org/doc/faq#Why_doesnt_Go_have_feature_X` (visited on 05/21/2020).

[84] R. Cox. (Feb. 2011). Tail call optimization - google groups, [Online]. Available: `https://groups.google.com/d/msg/golang-nuts/nOS2FEiIAaM/miAg83qEn-AJ` (visited on 06/05/2020).

[85] J. B. Snyder. (Oct. 2016). cmb/compile: improve inlining cost model, [Online]. Available: `https://github.com/golang/go/issues/17566`.

[86]   A. Gallant and contributors. (Mar. 2019). BurntSushi/go-sumtype, [Online]. Available: `https://github.com/BurntSushi/go-sumtype/tree/fcb4a6205bdc6ce526`
`f359ae5eae5fb6ded53916` (visited on 05/30/2020).

[87]   R. Rüttimann. (2020). ruettram/bachelor, ZHAW, [Online]. Available: `https://gi`
`thub.com/tommyknows/bachelor-thesis` (visited on 03/13/2020).

List of source codes

List of Figures

List of Tables

Glossary

AST Abstract Syntax Tree, an abstract representation of source code as a tree. 16, 36, 62, 63

copy by value Copy by value refers to the argument-passing style where the supplied argumens are copied. To pass references in Go, the developer needs to use pointers instead. 27

DAG Directed Acyclic Graph. 16

EBNF Extended Backus-Naur Form, an extended version of the 'Backus-Naur Form, a notation technique to describe programming language syntax. 33

SSA Single Static Assignment, an intermediate representation between the AST and the compiled binary that simplifies and improves compiler optimisations. 16, 36

stdout Standard Output, the default output stream for programs. 54

sum types Sum types, often also called aggregated types, variant or tagged union is a data structure that can hold one of several, predefined data types. For example, Haskell's `Either` holds either a value of type A or type B. Similar to that, `Maybe` can hold either a concrete value, or 'Nothing'.. 66

Appendices

1 Information about this thesis

This thesis, including this document, is contained in a single git repository that can be found at Github.

All the work that has been done is open sourced under the Apache License 2.0, except the Go source code, which has its own license.

To view the code, build or reproduce results, this git repository can be cloned and its submodules checked out. The version that has been submissed can be checked out through the tag 'v1.0.0':

```
$> git clone https://github.com/tommyknows/bachelor-thesis.git
$> cd bachelor-thesis
$> git checkout v1.0.0
$> git submodule init
$> git submodule update
```

The directory 'thesis' contains the LaTeX code for this paper. There is a helper required for displaying code in the thesis which is located in 'thesis/utils' and needs to be built with 'go build .'. However, a PDF which should be on the same state as the source is provided in the repository too ('thesis/thesis.pdf').

The 'work' directory contains code that has been developed in this thesis. In this directory, the Go and funcheck source code can be found (see Appendix 9 and 10). Further, some examples of functional Go have been developed in the 'example' directory. The 'common-list-functions' folder contains the script as mentioned in Appendix 3.

2 Example for Functional Options

This source code example demonstrates 'functional options' as introduced in this paper. Functional options are usually passed to a constructor to configure the new instance, in this case a webserver. The advantages of functional options are that the API ends up to be cleaner and more easily extensible and allows the default use-case to be as simple as possible.

```
1  package webserver

   type Server struct {
```

```go
      Timeout time.Duration
5     Port int
      ListenAddress string
}
type Option func(*Server)

10 func Timeout(d time.Duration) Option {
      return func(s *Server) { s.Timeout = d }
}
func Port(p int) Option {
      return func(s *Server) { s.Port = p }
15 }

func New(opts ...Option) *Server {
      s := &Server{ // initialise with default values
          Timeout: 500*time.Millisecond,
20        Port: 0, // uses a random port on the host
          ListenAddress: "http://localhost",
      }
      // apply all options
      for _, opt := range opts {
25        opt(s)
      }
      return s
}

30 // usage examples from outside the package:
s := webserver.New() // uses all the default options
s := webserver.New(webserver.Timeout(time.Second), webserver.Port(8080))
```

Source Code 1: Functional Options for a simple Webserver

3 Analysis of function occurrences in Haskell code

The results of the analysis have been aquired by running the 'count-function' script that is located in 'work/common-list-functions' in the git repository[87].

The script utilises ripgrep to count the number of occurrences, so this must be installed in order to run this script.

```
$> ./count-function.sh ": " "map " fold filter take drop reverse sum zip
↪  product maximum minimum
Searching for occurrences in subdirectories of
↪  bachelor-thesis/work/common-list-functions
Found 2912 occurrences of :
Found 1241 occurrences of map
Found 610 occurrences of fold
Found 562 occurrences of filter
Found 376 occurrences of take
Found 238 occurrences of drop
Found 185 occurrences of reverse
Found 173 occurrences of sum
Found 158 occurrences of zip
Found 56 occurrences of product
Found 53 occurrences of maximum
Found 26 occurrences of minimum
```

4 Mutating variables in Go

```go
package main

import "fmt"

func main() {
  m := MyStruct{
    x: "struct",
    y: 42,
  }

  fmt.Println(m) // {struct 42}
  mutateNoPointer(m)
  fmt.Println(m) // {struct 42}
  mutatePointer(&m)
  fmt.Println(m) // {changed 0}
}
```

```
     type MyStruct struct {
       x string
20     y int
     }

     func mutateNoPointer(m MyStruct) {
       m.x = "changed"
25     m.y = 0
     }
     func mutatePointer(m *MyStruct) {
       m.x = "changed"
       m.y = 0
30   }
```

Source Code 2: Example on how to mutate complex types in Go

5 Shadowing variables in Go

```
1  package main

   import "fmt"

5  func main() {
     x := 5
     fmt.Println(x) // 5
     // introducing a new scope
     {
10     // this assignment would be forbidden, as it
       // overwrites the parent block's value value.
       x = 3
       fmt.Println(x) // 3
     }
15   fmt.Println(x) // 3
     // introducing a new scope
     {
       // this redeclares the variable x, effectively
```

```
      // shadowing it. This will not change the parent
20    // block's variable.
      x := 4
      fmt.Println(x) // 4
    }
    fmt.Println(x) // 3
25  }
```

Source Code 3: Example on how shadowing works on block scopes

6 Foldl and Foldl' difference

This code example shows the difference between foldl and foldl' in their execution. What can be seen is that foldl builds up a call stack, while foldl' executes the calls during the traversal.

```
> (?) :: Int -> Int -> Int
> _ ? 0 = 0
> x ? y = x*y
>
> list :: [Int]
> list = [2,3,undefined,5,0]
>
> foldl (?) 1 list
foldl (?) 1 [2,3,undefined,5,0] -->
foldl (?) (1 ? 2) [3,undefined,5,0] -->
foldl (?) ((1 ? 2) ? 3) [undefined,5,0] -->
foldl (?) (((1 ? 2) ? 3) ? undefined) [5,0] -->
foldl (?) ((((1 ? 2) ? 3) ? undefined) ? 5) [0] -->
foldl (?) (((((1 ? 2) ? 3) ? undefined) ? 5) ? 0) [] -->
(((((1 ? 2) ? 3) ? undefined) ? 5) ? 0 -->
0

> foldl' (?) 1 list
foldl' (?) 1 [2,3,undefined,5,0] -->
    1 ? 2 --> 2
foldl' (?) 2 [3,undefined,5,0] -->
```

```
    2 ? 3 --> 6
foldl' (?) 6 [undefined,5,0] -->
    6 ? undefined -->
*** Exception: Prelude.undefined
```

Source Code 4: foldl and foldl' strictness[56]

7 Workaround for the missing foldl implementation in Go

This code block exemplifies how foldl could be implemented in Go code. It is based on the example from Appendix 6, rewritten in Go. While Go does not know about the concept of laziness, the programmer may implement the laziness himself by working with function closures.

In this example, *int is used instead of int to simulate Haskell's undefined with a nil-pointer. If a nil-pointer is dereferenced, the program will panic.

In the lazy version of this code (utilising 'myFold' and 'mulLazy'), the panic does not occur because the nil-pointer is never dereferenced as the function closure is never executed. This is equal to the 'foldl' demonstration in Appendix 6.

The non-lazy version ('foldl' and 'mul') executes the function while traversing the slice and thus panics.

```go
1   package main

    import "fmt"

5   func main() {
      zero, one, two, three := 0, 1, 2, 3
      list := []*int{&one, &two, nil, &three, &zero}

      // this will work, as the values will be evaluated
10    // "lazily" - the nested functions will never
      // be executed, thus it will never panic.
      fmt.Printf("%v\n", myFold(mulLazy, 1, list))
      // This will panic.
      fmt.Printf("%v\n", foldl(mul, 1, list))
15  }
```

```
    func mul(x int, y *int) int {
      if *y == 0 {
        return 0
20    }
      return x * *y
    }

    func mulLazy(x func() int, y *int) func() int {
25    return func() int {
        if *y == 0 {
          return 0
        }
        return x() * *y
30    }
    }

    func myFold(f func(func() int, *int) func() int, acc int, list []*int)
    ↪   int {
      a := func() int { return acc }
35    a = foldl(f, a, list)
      return a()
    }
```

```
$> fgo run .
0
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x109e945]

goroutine 1 [running]:
main.what(0x2, 0x0, 0x2)
    /tmp/map/main.go:16 +0x5
main.main()
    /tmp/map/main.go:12 +0x187
exit status 2
```

Source Code 5: Working around the missing foldl implementation in Go

8 Prettyprint implementation

These code blocks show the same package 'prettyprint', once in idiomatic Go and once in functional Go. What can be seen is that the `for` loops have been replaced by the usage of 'foldl' and anonymous functions.

```
1   /*
    The prettyprint package visualises a Go AST, but only assignment
    nodes. In these nodes, it prints their position, the identifier
    and the identifier's declaration plus position.

5
    It is used to visualise how assigncheck is checking for re-assignments.

    To see it for yourself, run `go test -v .` in this directory.

10      $> go test -v .
    === RUN    TestRun
    Assignment "x, y := 1, 2": 2958101
            Ident "x": 2958101
                    Decl "x, y := 1, 2": 2958101
15          Ident "y": 2958104
                    Decl "x, y := 1, 2": 2958101
    Assignment "y = 3": 2958115
            Ident "y": 2958115
                    Decl "x, y := 1, 2": 2958101
20      --- PASS: TestRun (0.93s)
    PASS
    ok      github.com/tommyknows/funcheck/prettyprint      1.088s
    */
    package prettyprint
25
    import (
      "bytes"
      "fmt"
      "go/ast"
30    "go/printer"
      "go/token"

      "golang.org/x/tools/go/analysis"
    )
```

```
35   var Analyzer = &analysis.Analyzer{
       Name: "prettyprint",
       Doc:  "prints positions",
       Run:  run,
40   }

     func run(pass *analysis.Pass) (interface{}, error) {
       for _, file := range pass.Files {
         ast.Inspect(file, func(n ast.Node) bool {
45         switch as := n.(type) {
           case *ast.DeclStmt:
             checkDecl(as, pass.Fset)
           case *ast.AssignStmt:
             checkAssign(as, pass.Fset)
50         }
           return true
         })
       }

55     return nil, nil
     }

     func checkDecl(as *ast.DeclStmt, fset *token.FileSet) {
       fmt.Printf("Declaration %q: %v\n", render(fset, as), as.Pos())
60     decl, ok := as.Decl.(*ast.GenDecl)
       if !ok {
         return
       }

65     for i := range decl.Specs {
         val, ok := decl.Specs[i].(*ast.ValueSpec)
         if !ok {
           continue
         }

70
         if val.Values != nil {
           continue
         }
```

```go
75      if _, ok := val.Type.(*ast.FuncType); !ok {
          continue
        }

        fmt.Printf("\tIdent %q: %v\n", render(fset, val),
        ↪ val.Names[0].Pos())
80    }
    }

    func checkAssign(as *ast.AssignStmt, fset *token.FileSet) {
      type pos interface {
85        Pos() token.Pos
      }

      fmt.Printf("Assignment %q: %v\n", render(fset, as), as.Pos())

90    for _, expr := range as.Lhs {
        ident := expr.(*ast.Ident) // Lhs always is an "IdentifierList"

        fmt.Printf("\tIdent %q: %v\n", ident.String(), ident.Pos())

95      // skip blank identifiers
        if ident.Name == "_" {
          fmt.Printf("\t\tBlank Identifier!\n")
          continue
        }
100
        if ident.Obj == nil {
          fmt.Printf("\t\tDecl is not in the same file!\n")
          continue
        }
105
        // make sure the declaration has a Pos func and get it
        declPos := ident.Obj.Decl.(pos).Pos()
        fmt.Printf("\t\tDecl %q: %v\n", render(fset, ident.Obj.Decl),
        ↪ declPos)
      }
110  }

    // render returns the pretty-print of the given node
```

```go
func render(fset *token.FileSet, x interface{}) string {
  var buf bytes.Buffer
  if err := printer.Fprint(&buf, fset, x); err != nil {
    panic(err)
  }
  return buf.String()
}
```

Source Code 6: The original prettyprint implementation

```go
package prettyprint

import (
  "bytes"
  "fmt"
  "go/ast"
  "go/printer"
  "go/token"

  "golang.org/x/tools/go/analysis"
)

var Analyzer = &analysis.Analyzer{
  Name: "prettyprint",
  Doc:  "prints positions",
  Run:  run,
}

type null struct{}

func checkDecl(as *ast.DeclStmt, fset *token.FileSet) {
  fmt.Printf("Declaration %q: %v\n", render(fset, as), as.Pos())

  check := func(_ null, spec ast.Spec) (n null) {
    val, ok := spec.(*ast.ValueSpec)
    if !ok {
      return
    }
    if val.Values != nil {
```

```go
      return
    }
    if _, ok := val.Type.(*ast.FuncType); !ok {
      return
    }
    fmt.Printf("\tIdent %q: %v\n", render(fset, val),
    ↪  val.Names[0].Pos())
    return
  }

  if decl, ok := as.Decl.(*ast.GenDecl); ok {
    _ = foldl(check, null{}, decl.Specs)
  }
}

func checkAssign(as *ast.AssignStmt, fset *token.FileSet) {
  fmt.Printf("Assignment %q: %v\n", render(fset, as), as.Pos())

  check := func(_ null, expr ast.Expr) (n null) {
    ident, ok := expr.(*ast.Ident) // Lhs always is an "IdentifierList"
    if !ok {
      return
    }

    fmt.Printf("\tIdent %q: %v\n", ident.String(), ident.Pos())

    switch {
    case ident.Name == "_":
      fmt.Printf("\t\tBlank Identifier!\n")
    case ident.Obj == nil:
      fmt.Printf("\t\tDecl is not in the same file!\n")
    default:
      // make sure the declaration has a Pos func and get it
      declPos := ident.Obj.Decl.(ast.Node).Pos()
      fmt.Printf("\t\tDecl %q: %v\n", render(fset, ident.Obj.Decl),
      ↪  declPos)
    }

    return
  }
```

```
  _ = foldl(check, null{}, as.Lhs)
}

func run(pass *analysis.Pass) (interface{}, error) {
  inspect := func(_ null, file *ast.File) (n null) {
    ast.Inspect(file, func(n ast.Node) bool {
      switch as := n.(type) {
      case *ast.DeclStmt:
        checkDecl(as, pass.Fset)
      case *ast.AssignStmt:
        checkAssign(as, pass.Fset)
      }
      return true
    })
    return
  }
  _ = foldl(inspect, null{}, pass.Files)

  return nil, nil
}

// render returns the pretty-print of the given node
func render(fset *token.FileSet, x interface{}) string {
  var buf bytes.Buffer
  if err := printer.Fprint(&buf, fset, x); err != nil {
    panic(err)
  }
  return buf.String()
}
```

Source Code 7: The refactored, functional prettyprint implementation

9 Compiling and using functional Go

To compile and use the changes to the Go compiler that have been implemented in this thesis, these instructions should be followed.

First, check out the Go source code:

```
$> git clone https://github.com/tommyknows/go.git
$> cd go
$> git checkout bachelor-thesis
```

9.1 With Docker

If Docker is installed on your system, you can follow these steps from within the checked out 'go' git repository on the branch 'bachelor-thesis'. The downside of this approach is that it complicates building and sharing binaries. To compile your own project, the directory has to be mounted into the container. If your Guest OS is not Linux, cross-compilation is required so that the executable can be ran on the host.

These steps have been wrapped inside a script that prints out the necessary commands to configure your environment.

```
$> eval "$(./setup-docker.sh)"
```

The commands within the shell script will build Go in the container and print commands to configure the environment. The 'eval' command then executes these printed commands. Executing this command may take a while, as the Go compiler is compiled within this process.

To build projects with the functional Go installation, simply use 'fgo' on the command line. An alias has been created that mounts the current directory and executes the 'fgo' command within the container.

Note that if this only configures the 'fgo' command in the current shell session. To persist it across shell-sessions, execute the script without eval:

```
$> ./setup-docker.sh
```

And add the printed commands to your '.bashrc' (or equivalent). Further, you may also need to change the binary path from '/tmp/fgo/bin' to a path which is not cleaned up regularly.

9.2 With a working Go installation

If you already have a working Go installation on your system, the following steps provide a way to get functional Go up and running in the same way a normal go installation does.

These steps need to be executed from within the checked out 'go' git repository on the branch 'bachelor-thesis'.

Build the functional Go binary and configure the environment:

```
$> cd ./src
$> ./make.bash
$> ln -s $(realpath $(pwd)/../bin/go) /usr/local/bin/fgo
$> go env -w GOROOT=$(realpath $(pwd)/..)
```

The 'go env' command sets the GOROOT to point to the newly compiled tools and source code and is valid for the current shell session only.

9.3 Using the installation

After these steps, the binary (or alias) 'fgo' can be used to test and build functional Go code. 'fgo' is not different to the normal 'go' command, so all commands that work with the normal 'go' command also work with the 'fgo' command.

```
$> cd <code directory>
$> fgo test ./...
$> fgo build ./...
```

10 Building Funcheck

Funcheck needs to be built against functional Go to properly detect the builtin functions. If you have not done so already, install 'fgo' as shown in Appendix 9.

Then, funcheck can be installed directly with 'go get' (or rather, 'fgo get'). 'go get' downloads the source code to the Go modules directory (usually in $GOPATH/pkg/mod), compiles the specified package and moves the binary to $GOPATH/bin.

```
$> # go get should not be called from within a go module
$> cd /tmp
$> fgo get github.com/tommyknows/funcheck
$> funcheck -h
```

This installs 'funcheck' into `$GOBIN` or, if `$GOBIN` is not set, into `$GOPATH`/bin.

You can also clone the git repository and use 'fgo build' to build 'funcheck':

```
$> git clone https://github.com/tommyknows/funcheck.git
$> cd funcheck
$> fgo build .
$> mv ./funcheck /usr/local/bin/funcheck
$> funcheck -h
```

To run funcheck against the current directory / package, simply run

```
$> funcheck .
```

11 Building Gopls

Gopls is the official language server for Go. Similar to funcheck, there are two options to install it on your local machine.

Installing with 'go get':

```
$> # go get should not be called from within a go module
$> cd /tmp
$> fgo get golang.org/x/tools/gopls
$> gopls -h
```

Or by downloading the source manually:

```
$> git clone https://github.com/golang/tools.git
$> cd ./tools/gopls
$> fgo build .
$> mv ./gopls /usr/local/bin/gopls
$> gopls -h
```

Anhang/Appendix:
  – Projektmanagement:
    – Offizielle Aufgabenstellung, Projektauftrag
    – (Zeitplan)
    – (Besprechungsprotokolle oder Journals)
  – Weiteres:
    – CD/USB-Stick mit dem vollständigen Bericht als PDF-File inklusive Film-
und Fotomaterial
    – (Schaltpläne und Ablaufschemata)
    – (Spezifikation u. Datenblätter der verwendeten Messgeräte und/oder Kompo-
nenten)
    – (Berechnungen, Messwerte, Simulationsresultate)
    – (Stoffdaten)
    – (Fehlerrechnungen mit Messunsicherheiten)
    – (Grafische Darstellungen, Fotos)
    – (Datenträger mit weiteren Daten (z. B. Software-Komponenten) inkl. Verzeich-
nis der auf diesem Datenträger abgelegten Dateien)
    – (Softwarecode)