Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

Bachelor thesis in Computer Science
Spring 2020

Functional Go

an easier introduction to functional programming

Authors	Ramon Rüttimann
Main supervisor	Dr. G. Burkert
Sub supervisor	Dr. K. Rege

Date	24.05.2020

Zurich University
of Applied Sciences

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.

Zurich University
of Applied Sciences

Summary

(in Deutsch)

Abstract

(in Englisch)

Preface

Stellt den persönlichen Bezug zur Arbeit dar und spricht Dank aus.

Contents

1 Introduction

1.1 Learning Functional Programming

In 2007, C# 3.0 was released. Two years later, Ryan Dhal published the initial version of NodeJS, eliminating JavaScript's ties to the browser and introducing it as a server-side programming language. In 2013, Java 8 was released. Within the same timeframe, Python has been rapidly growing in popularity[1].

What all these events have in common is that they brought along concepts from functional programming, so far mainly used in academia, into the daily life of many programmers.

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.

Functional programming has landed in the wider programming-community, emerging from niche use-cases and academia. For example Rust, the 'most popular programming language' for 4 years in a row (2016–2019) according to the StackOverflow Developer survey[2], has been significantly influenced by functional programming languages[3]. 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[4][5][6][7], even if one may not end up using that language at all[8].

Even though the exact definition of what a *purely* functional language is remains a controversy[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 lazely-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
```

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 paranthesised arguments and no clear indicator at which arguments it takes and which types are returned

Though some of these points are also available to programmers in imperative or object-oriented languages, the cumulative difference is not to underestimate and adds to Haskell's steep learning curve.

1.3 Goals

Learning a new paradigm and syntax at the same time can be daunting and discouraging for novices. 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 ease the learning curve of functional programming, this thesis will consist of two parts. In the first part, writing functional code should be made as easy as possible. This means that a language with an easy and familiar syntax should be chosen. Further, this programming 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 is created to check code for non-functional statements. To achieve this, functional purity has to be defined, 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 only of 25 keywords and purposefully omits constructs like the ternary operator (<bool> ? <then> : <else>) as a replacement for the longer 'if <bool> { <then> } else { <else> }' due to 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 years of experience with the language.

There are however a few downsides of using Go. So far, 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 goes into more details on slices.

1.5 Existing Work

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

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

is used to register functions for http server handling:

```
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)
}
```

[19]

Using functions as function parameters or return types is a commonly used feature in Go, not just within the standard library.

1.5.1 Functional Options

A software design pattern that has gained popularity within the Go community is 'functional options'. The pattern has been outlined in Dave Cheney's blog post 'Functional options for friendly APIs' 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

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

  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:

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

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

```
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 1.

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 ported with Go. Ignoring speed and stackoverflows due to non-existent tail call optimisation[22], the main issue was with the type system and the missing polymorphism.

1.5.3 go-functional

In July 2017, Aaron Schlesinger, a Go programmer for Micosoft Azure, gave a talk on functional programming wit 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 Conclusion

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, a list of rules for pure functional should be curated and implmented in a

linter. This linter can then be used to check existing code and report constructs which are not functional.

2 Basics

2.1 Go Slices

As mentioned in section 1.4, Go does not have a 'list' implementation. 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 instead added 'slices' as a type to the language.

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.[25]*

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

```
// 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
    // 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 $<T>$.

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[26].

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.

However, what is missing from Go's slices are a lot of the classical list 'helper' functions. In a typical program written in a functional language, lists take a central role[1]. Because of this, functional languages have a number of helper functions like 'map', 'filter' and 'fold'[28] that currently do not exist in Go and would need to be implemented by the programmer. With no support for polymorphism, an 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:

```
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 these base types. Counting the different numeric types into that equation (totally 19 distinct types[29]), would grow that number to $19^2 = 361$ functions.

Though this code could be generated, it leaves out custom user-defined types, which would still need to be generated separately.

Another option would be that 'map' would take and return empty interfaces (`interface{}`). However, 'the empty interface says nothing'[30]. The declaration of 'map' would be:

```
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, converting slices between types cannot be done directly. Instead, a new slice has to be created:

[1] 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)[27]

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

This by itself is a usage pattern that cannot be justified. Further, the example 'map' function declaration does not specify which types must be equal; this would need to be a convention.

To mitigate this point, the most common list-operations (in Go slice-operations) will need to be added as built-ins to the compiler, so that the programmer can use these functions on every slice-type without any necessary conversions.

2.2 Built-in functions

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.[31]*

As Go does not have generics, functions that return a variable, but concrete type have to be built into the language itself.

For example, `append`:

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

This shows that the types supplied to append are not specified upfront. Instead, they are resolved 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[32], and not it's implementation. As of Go 1.14, there are two major implementations of that specification; Google's self-hosting compiler toolchain[2] 'gc', which is written in Go, and 'gccgo', a frontend for GCC, writen 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[33]. 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.[34]*

This is why Go has taken some measures to to combat slow compilation times. According to Rob Pike, one of the creators of Go, Go's compiler is not notably fast, but most other compilers are slow:

> *The compiler hasn't even been properly tuned for speed. A truly fast compiler would generate the same quality code much faster.[35]*

The language design does have some limitations however to lower 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'[35].

[2] This is what most often is referred to as 'the Go compiler'.
[3] 'gc' stands for 'Go Compiler', and not 'Garbage Collection' ('GC').

Compiling Go programs with the standard 'gc' compiler can be 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 tokenizing the code ('lexical analysis' - the 'lexer'), parsing ('syntax analysis' - the 'parser') it and then constructing a syntax tree (AST) for each source file[4].

In comparison to most production-level languages, Go can be parsed without a symbol table. It has been designed to be easy to parse and analyse, which makes the Go parser simple in it's design[37].

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 the 'gc's AST definition was carried over from the C implemenation.

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

After type-checking, some transformations are applied on the AST. This includes, but is not limited to, eliminating dead code, inlining function calls and escape analysis. What is also done in the transformation phase is rewriting builtin function calls, replacing for example a call to the builtin 'append' with the necessary AST structure and rutime-calls to implement its functionality.

2.3.3 SSA

In the third phase, the AST is converted to SSA 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'[38].

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 (e.g. replace expressions with constant

[4]Technically, the syntax tree is a syntax DAG[36]

values and optimise 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, including, but not limited to:

- Rewriting generic values into their machine-specific variants (for example, on amd64, combining load-store operations)
- Another dead-code elimination pass
- 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. [38]

3 Methodology

3.1 Slice Helper Functions

3.1.1 Choosing the functions

The first task is to implement some helper functions for slices, as they are present 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[39]:

- Shellcheck (koalaman/shellcheck[40]): A static analysis tool for shell scripts
- Pandoc (jgm/pandoc[41]): A universal markup converter
- Postgrest (PostgREST/postgrest[42]): REST API for any Postgres database
- Semantic (github/semantic[43]): Parsing, analyzing, and comparing source code across many languages
- Purescript (purescript/purescript[44]): A strongly-typed language that compiles to JavaScript
- Compiler (elm/compiler[45]): Compiler for Elm, a functional language for reliable webapps
- Haxl (facebook/haxl[46]): 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 have been counted. The analysis does not differentiate between different kind of functions. For example, 'fold' includes all occurrences of 'foldr', 'foldl' and 'foldl'́. 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.

Table 3.1: Occurrences of list functions2

map	1241
':' (cons)	1177
fold	610
filter	262
reverse	154
take	104
drop	81
maximum	53
sum	44
zip	38
product	15
minimum	10
reduce	8

1 2

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

3.1.2 Map

The most used function in Haskell is map. The table 3.1 counts 1200 occurrences of map, although 600 of those occurrences are from fmap. fmap is the generic version of map:

[1] 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.

[2] The map function listed here also includes occurrences of 'fmap'. A more detailed look at the data shows that there are 632 occurrences of 'fmap', which means that 'fmap' and 'map' are used equally as often. As 'fmap' however requires some kind of implementation for an 'iterable', it is out of scope for this paper.

while map only works on lists, fmap works on every type that implements the 'Functor' typeclass[3], including tuples, 'Maybe' and lists. 'In practice a functor represents a type that can be mapped over.'[47] A common analogy for functors are 'boxes'. A functor behaves like a box where a value can be put into and taken out again. For lists, the operation of 'taking out' elements means processing each value one by one. For 'Maybe', it means 'unpacking' the concrete value and processing it, and if there is no concrete value, returning 'Nothing' instead.

map (and fmap) is one of the most useful higher-order functions:

map

> *returns a list constructed by appling a function (the first argument) to all items in a list passed as the second argument[48].*

Some usage examples of map can be found at 3.1.

Source Code 3.1: Example usage for map and fmap

```
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"]
```

Due to missing polymorphism, map cannot be implemented as 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 types `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 builtin 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 'iterator'-like concept. The 'range' clause, used

[3] typeclasses in Haskell are what would be interfaces in imperative and object-oriented languages

with ‘for’ loops, only works on slices and maps - both are builtin data types. It does not work on custom data structures, which would make the implementation a far bigger task. 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.

Source Code 3.2: Example usage of map in go

```
package main

import (
  "fmt"
  "strconv"
)

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

[4]

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 ‘contents of the address register’), and the ‘cdr’ (‘content of decrement register’). 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[49].

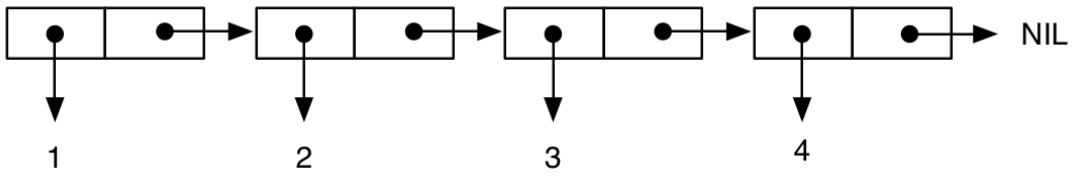

Figure 3.1: Cons cells forming a list[50]

> *An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.[51]*

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, to give an example:

Source Code 3.3: Example usage of prepend in go

```
package main

import (
  "fmt"
  "strconv"
)

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

3.1.4 Fold

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

> *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.[52]*

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:

Source Code 3.4: Function headers of the fold functions

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

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.

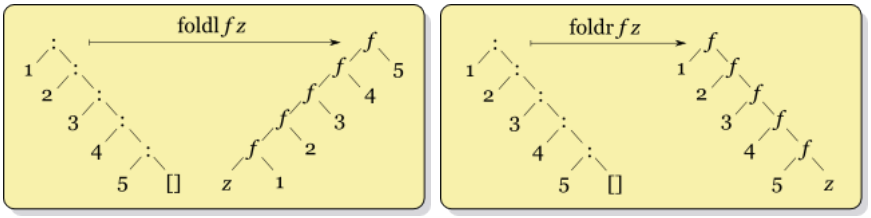

Figure 3.2: Folds illustrated[52]

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

Source Code 3.5: foldr and foldl execution order

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

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.[53]*

The strictness property is only relevant if the function is lazy in its first argument, meaning that foldl builds up an execution path, while foldl' executes the instructions while traversing it:

Source Code 3.6: foldl and foldl' strictness[53]

```
> (?) :: 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
```

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'.[5] The usage of these fold-functions is equal to the Haskell versions, where foldl's arguments are switched in order.

Source Code 3.7: Example usage of foldr and foldl in go

[5]If the behaviour from the normal foldl function is required, a workaround can be applied in th Go version. See appendix 5

```
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
}
```

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:

Source Code 3.8: Example usage of filter in 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]
}
```

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'[6], with different 'rules' that are reported upon. The name for this tool will be 'funcheck' because functional programming should be fun.

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[9], 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 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.[55]*

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 unchangable. This means that their state cannot be modified after it's creation and first assignment.

Many programming languages provide features for setting variables as immutable.

[6] A list of Go linters can be found in golangci-lint[54].

In Rust for example, variables are immutable by default. To explicitely 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[56]*

'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[7]. A complex type — struct, interface, function — cannot be constant.

This means that the programmer cannot write `const x = MyStruct{A: a, B: b}` to make a struct immutable, as a struct is a complex type. Therefore, the solution to making variables immutable is to explicitely not allow any mutations in a program[8]. While this incorporates a lot of different aspects, the simplest solution to that 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].

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

[8] Although this may not be a very strict definition of immutability and doesn't allow for compiler optimisations or similar, it is enough in the context of learning functional programming and learning that variables cannot be modified.

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

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

Go being a copy by value language, mutating existing variables accross functions works by passing pointers to these variables instead[10]. When removing the regular assignment operators, the usage of pointers becomes second nature. Technically, they could be used everywhere now, as it is not possible to mutate anything either way. It is left to the developer to decide, but it is not necessary to use pointers anymore. Functional languages, in contrast, do not provide the option to address variables, since this is merely an optimisation that is left to the compiler.

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 4.

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 evaluation of the function is also not allowed to have any 'side effects', meaning that it should not mutate non-local variables or references[11].

As discussed in the last Chapter 3.2.2, if re-assignments are not allowed, mutating global variables is not possible. The same logic can be followed with variables passed by reference (pointers).

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

As such, forbidding re-assignments is an extremely simple solution to ensure functional purity within the program. However, this solution misses an important aspect in most programs: 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.

[10] An example for this can be found at 3.
[11] This also includes 'local static' variables, but Go does not have a way to make variables 'static'.

Input and Output in a program — through wether channel it may be — is impure by design. Due to this, the Haskell language auhtor 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. Becasue of this, the execution order of functions is not rellevant, 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, omitting calls if their return values are unused or lazily evaluating functions.

However, impure IO functions need to be considered too.

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"`. As there is a magical computer that never files, this function does not return anything; it always succeeds.

Because of this, the function does not have any return value. Or it may return a value for the bytes written, but that value 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 (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[12]:

```
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'[57].

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. It enables Haskellers to use impure functions in anotherwise 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.

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, 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 re-assignments in the code. There are a few exceptions, 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.

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

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:

Source Code 3.9: Go Variable Declarations

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

What can be seen in this example is that the type of the variable can be deducted automatically[13], or be specified explicitely.

However, the notation `var k = 1` is seldomely 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:

```
k := 1
```

> *It is shorthand for a regular variable declaration with initializer expressions but no types:*
>
> ```
> "var" IdentifierList = ExpressionList .
> ```

[58]

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 it is less verbose than its counterpart `var x = y`.

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

Source Code 3.10: Go Assignment Operators

[13] 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`.

```
var x = 5
x += 5 // equal to x = x + 5
x %= 3 // equal to x = x % 3
x <<= 2 // equal to x = x << 2 (bitshift)
```

14

All these operators are re-assigning values and are not available in functional programming.

However, there are a few exceptions to this rule, which are covered in the following chapters.

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. [60]*

This definition holds an important caveat for function definitions:

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

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

Source Code 3.11: Go scoping issue with recursive functions

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

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 at the closing brace of function f.

[14] Non-exhaustive list of operators. A complete listing of operators can be found in the Go language spec[59].

This cannot be changed within the compiler without touching a lot of the scoping rules. Changing these scoping rules may have unintended side effects. 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:

Source Code 3.12: Fixing the scope issue on recursive functions

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

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 Extended Backus-Naur Form 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[15].

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[16] variables is new. As a consequence, redeclaration*

[15] The `var` keyword also takes lists, so this is possible too
[16] The blank identifier (`_`) discards a value; `x, _ := f()`

> *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****.*
>
> ```
> field1, offset := nextField(str, 0)
> field2, offset := nextField(str, offset)  // redeclares offset
> ```
>
> *[58]*

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 aformentioned 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:

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

The while loop:

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

The regular C-style for loop:

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

And the range loop, which allows to range (iterate) over slices and maps[17]:

[17] And channels, which are not touched upon in this thesis

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

Range returns two values, both of which can be omitted[18]. When ranging over a slice, the index and a copy of the element at that index is returned. For maps, a 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 funcheck 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 an (internal) reassignment.

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, it is easier for the user if he gets a notification on the loop construct instead.

The infinite loop does not contain any reassignments and would thereby be legal by the definition of the reassignment rule. However, in purely functional programming language, loops do only exist as a concept, and are implemented with, for example, fold's and maps.

For these reasons, for loops will be reported in funcheck.

[18]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` is equal to just `i := range s`.

4 Implementation

4.1 Implementing the new built-in functions

4.1.1 Required Steps

Adding a builtin 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[61] 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 for that node type, translating it to AST nodes that the compiler already knows and can translate to builtin 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 builtins.

The 'private' implementation is located within the *src/cmd/compile/internal* package[63]. It can only be used by the package in *src/cmd/compile*, which contains the implementation of the compiler itself.

When calling `go build .`, the compiler is invoked indirectly through the main 'go' binary. To directly invoke the compiler, `go tool compile` can be used.

[1] Gopls is Go's official language server implementation[62].

[2] "An import of a path containing the element "internal" is disallowed if the importing code is outside the tree rooted at the parent of the 'internal' directory."[63]

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 [61]. This also applies to builtin functions in the compiler, which have a function stub to document their behaviour[64], but no implementation, as that is done in the compiler[65].

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

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

```
135 // ...
    // The prepend built-in function prepends an element to the start of
    // the slice.  As slices do not have any capacity in the beginning,
    // this function always results in an expensive copy of the original
    // slice. Prepend returns the updated slice. It is therefore necessary
140 // to store the result of prepend, often in the 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
145 // a slice of elements of another type, using the given function.
    // The returned slice always has the same number of elements as the
    // source slice.
    func fmap(fn func(Type) Type1, slice []Type) []Type1

150 // The fold built-in functions fold over a slice of elements with the
    // given function. It takes init, the second argument, and the 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
155 // at the last, foldl at the first element of the slice.
    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
// element will be added to the returned slice.
// The returned slice will always have the same capacity as the
// original slice, but the length will always be equal to the
// number of elements that returned true in the filter function.
func filter(fn func(Type) bool, slice []Type) []Type

// ...
```

4.1.3 Public packages

> *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'.[38]*

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 builtins is 'go/types'.

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

Source Code 4.2: Registering new built-in functions

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

const (
  // universe scope
  _Append builtinId = iota
  _Prepend
  _Fmap
  _Foldr
  _Foldl
```

```
  _Filter
  // ...
```

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

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 are to be implemented.

This concludes the type-checking for external tools and makes 'gopls' return errors Once type-checking the new built-in functions is implemented, 'gopls' can be compiled against the new public packages.[3] It 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"

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

[3]This means pointing the go toolchain to the correct directory by setting the value of 'GOROOT' with `go env -w GOROOT=<path>`.

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. It should also check the node's child nodes, meaning an operations arguments, body and init statements. 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 now 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, which is why these steps are elided in this paper.

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

[4]The full implementations can be viewed by diff-ing the git repository between the references 'bachelor-thesis' and 'go1.14'[66].

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:

Source Code 4.3: Fmap implementation in Go

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

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.

Source Code 4.4: Improved implementation of fmap

```
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
}
```

This algorithm can be translated to the following AST node:

Source Code 4.5: fmap AST translation[67]

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

```
//
func walkfmap(n *Node, init *Nodes) *Node {
  // ...
```

prepend

The general algorithm for 'prepend' is:

Source Code 4.6: prepend implementation in Go

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

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:

Source Code 4.7: prepend AST translation[68]

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

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:

Source Code 4.8: fold implementation in 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
}
```

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:

Source Code 4.9: fold AST translation[69]

```
// ...
// walkfold rewrites the builtin fold function.
// For the right fold:
//   foldr(f(T1, T2) T2, a T2, s []T1) T2
//
//     init {
//       acc = a
```

```
//       for i := len(s) - 1; i >= 0; i-- {
//         acc = f(s[i], acc)
//       }
//     }
//     acc
//
// And the left fold:
//   foldl(f(T2, T1) T2, a T2, s []T1) T2
//
//     init {
//       acc = a
//       for i := 0; i < len(s); i++ {
//         acc = f(acc, s[i])
//       }
//     }
//     acc
func walkfold(n *Node, init *Nodes, isRight bool) *Node {
  // ...
```

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.

Source Code 4.10: filter implementation in Go

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

And the same algorithm, but translated to AST statements:

Source Code 4.11: filter AST translation[70]

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

4.2 Functional Check

As discussed in Chapter 3.2, a linter needs to be written to detect reassignments within a Go program.

To get a grasp about the issues this linter is trying to report, the first step is to capture 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 statement `x = 6` and `y = 6` 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:

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

All the aforementioned examples and more can be found in the testcases for funcheck[71].

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[72]. With this package, implementing a static code analyizer is being reduced to writing the actual AST node analysis.

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

```
var Analyzer = &analysis.Analyzer{
  Name: "assigncheck",
  Doc:  "reports re-assignments",
  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:

```
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. The basic building blocks for our is the following `switch` statement:

```
switch as := n.(type) {
case *ast.IncDecStmt:
  pass.Reportf(as.Pos(), "inline re-assignment of %s", as.X)

case *ast.ForStmt:
  pass.Reportf(as.Pos(), "internal reassignment (for loop) in %q",
   ↪  renderFor(pass.Fset, as))

case *ast.RangeStmt:
  pass.Reportf(as.Pos(), "internal reassignment (for loop) in %q",
   ↪  renderRange(pass.Fset, as))
}
```

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`:

```
// An IndexExpr node represents an expression followed by an index.
IndexExpr struct {
  X      Expr      // expression
```

[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`.

```
  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.

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 4.12. 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.

Source Code 4.12: Illustration of an assignment node and corresponding positions[74]

```
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
```

[7] 'An object describes a named language entity such as a package, constant, type, variable, function (incl. methods), or a label'[73].

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

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 4.12).

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.

The first step is to detect function declarations; statements of the form `var f func()`. Should such a statement be encountered, its position will be 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 4.13

Source Code 4.13: Illustration of a function literal assignment[74]

```
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
```

With this technique it is possible to excempt functions from the no-reassignment rule.

4.2.5 Testing Funcheck

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

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

Source Code 4.14: Testing a code analyser with the 'analysistest' package

```
package assigncheck

import (
        "testing"

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

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

The library expects the testdata 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 Refactoring the Prettyprint Package

The code blocks 4.12 and 4.13 have been generated by a small package 'prettyprint' contained in the funcheck repository.

To see how the newly bult-in functions and funcheck can be used, we refactor 'prettyprint' 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.

Similarily 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` (declarations) and `*ast.AssignStmt` (assignments).

For example, for the program

```
package main

import "fmt"

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

the following AST information is printed:

[1] Although there is no exact definition of what idiomatic Go is, so this interpretation could be challenged. It is idiomatic Go code to the author of this thesis.
[2] Assigncheck is the main package for funcheck and checks the reassignments

```
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 statements that are not functional:

```
$> funcheck .
prettyprint.go:20:2: internal reassignment (for loop) in
↪  "for _, file := range pass.Files { ... }"
prettyprint.go:30:5: internal reassignment (for loop) in
↪  "for i := range decl.Specs { ... }"
prettyprint.go:53:5: 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 re-assignments of variables in the code.

The code to print declarations is as shown in 5.1.

Source Code 5.1: Pretty-printing declarations in idiomatic Go

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

  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(pass.Fset, val),
    ↪  val.Names[0].Pos())
  }
}
```

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

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

As 'foldl' requires a return type, we introduce a dummy type 'null', which is just an empty struct:

```
type null struct{}
```

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

```
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 snippet 5.1 thus can be translated to

Source Code 5.2: Pretty-printing declarations in functional 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)
  }
}
```

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

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 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.

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

5.2 Quicksort

In Chapter 1.2, 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:

Source Code 5.3: Quicksort Implementations compared

```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
```

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.

5.3 Comparison to Java Streams

In Java 8, concepts from functional programing 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.

It is similar to the new built-in functions in this thesis:

Source Code 5.4: Comparision Java Streams and Functional Go

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

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

The lambda-syntax in Java is more concise than Go's function literals, where the complete function header has to be provided[4]. Java also has a more verbose way to declare anonymous functions, which is creating an anonymous class:

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

[76]

If this is used, Go's function literal syntax has an edge over Java's when it comes to readability and conciseness.

[4]There is an open proposal to add a lightweight anonymous function syntax to Go 2, which, if implemented, would resolve the verbosity[75]

Furthermore, the conversion to a stream and back to a list (`list.stream()` and `.collect(Collectors.`
is not required in Go, as the operations all work on slices. Here, only having a single list-like type built into the language is an advantage, as the (at least syntactical) overhead to convert the list only to run a 'filter' function can be avoided.

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

However, 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 is more than 700 pages[5], more than 6 times the size.

[5]The Java 8 Specification is 724[77], the Java 14 Specification 774[78] pages.

6 Experiments and 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 is purely functional or if there are still 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. While the IO monad does not make impure functions pure, it does serve as documentation to it's 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. 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:

```
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 ran 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 on arrays, mitigate a lot of downsides when comparing regular arrays to lists[1].

What Go's slices lack on the other side 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, we analysed the most popular open-source Haskell projects (pandoc, shellcheck and purescript, to name a

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

few). 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 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 more effort. 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 compiling phase, as well as a polymorphic function header, allowing the programmer to call the function with any allowed type.

A demonstration of these functions and how functional Go code looks like can be found at 6.1

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.

[2]If the language server (gopls) is compiled against the modified version of Go9

Source Code 6.1: Demonstration of the new built-in functions

```
package main

import (
  "fmt"
  "strconv"
)

type parity bool

const (
  even parity = true
  odd  parity = false
)

// 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 == 0) == p
  }
}

func main() {
  l := []int{1, 2, 3, 4, 5}
  l5 := fmap(func(i int) int { return i * 5 }, prepend(0, l))

  // fold over even / odd numbers and add them to a string
  evens := foldl(
    func(s string, i int) string { return s + strconv.Itoa(i) + " " },
    "even: ",
    filter(shouldBe(even), l5),
  )
  odds := foldl(
    func(s string, i int) string { return s + strconv.Itoa(i) + " " },
    "odd: ",
    filter(shouldBe(odd), l5),
  )

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

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, we argue that, especially for the first steps in functional programming,

> *Clear is better than clever[79]*

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, we 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 implicit 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

[1] For example, Go 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.

in order to keep the complexity of code to a minimum[80]. The simplicity of the 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]. Regardless, the performance of a language is not as important if it is not used in a production environment, which is why it was never a criteria in the first place.

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

Bibliography

[1] 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).

[2] '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).

[3] R. C. Steve Klabnik Carol Nichols. (2018). Functional language features, [Online]. Available: https://doc.rust-lang.org/book/ch13-00-functional-features.html.

[4] 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).

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

[6] 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).

[7] 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).

[8] (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).

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

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

[11] (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] (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] (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] (Sep. 2007). The Haskell learning curve, [Online]. Available: https://wiki.haskell.org/index.php?title=Humor/LearningCurve&oldid=15543 (visited on 03/04/2020).

[16] (Feb. 2020). Introduction - haskell wiki, [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] (). Frequently Asked Questions, [Online]. Available: https://golang.org/doc/faq#Does_Go_have_a_ternary_form (visited on 05/21/2020).

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

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

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

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

[27] Tikhon 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).

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

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

[30] Rob 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).

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

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

[33] (Jun. 2012). Less is exponentially more, [Online]. Available: https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html (visited on 04/10/2020).

[34] (). Frequently Asked Questions, [Online]. Available: https://golang.org/doc/faq#creating_a_new_language (visited on 04/10/2020).

[35] (Mar. 2014). Why does go programs compile faster than java or c#? [Online]. Available: https://groups.google.com/d/msg/golang-nuts/al4iuFXLPeA/PYNcUQ0_uAEJ (visited on 04/10/2020).

[36] (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).

[37] (). Frequently Asked Questions, [Online]. Available: https://golang.org/doc/faq#different_syntax (visited on 04/10/2020).

[38] (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).

[39] (2020). Search - stars:>1000 language:Haskell, Github, [Online]. Available: https://github.com/search?l=&o=desc&q=stars%3A%3E1000+language%3AHaskell&s=stars&type=Repositories (visited on 03/12/2020).

[40] (Mar. 2020). koalaman/shellcheck: ShellCheck, a static analysis tool for shell scripts, Github, [Online]. Available: https://github.com/koalaman/shellcheck/tree/68a03e05e5e030d7274c712ffa39768310e70f8a (visited on 03/12/2020).

[41] (Mar. 2020). jgm/pandoc: Universal Markup Converter, Github, [Online]. Available: https://github.com/jgm/pandoc/tree/91f2bcfe73fa3a489654ee74bca02e24423dc5c0 (visited on 03/12/2020).

[42] (Mar. 2020). PostgREST/postgrest: REST API for any Postgres database, Github, [Online]. Available: https://github.com/PostgREST/postgrest/tree/dea57bd1bec9ba5c4e43 (visited on 03/12/2020).

[43] (Mar. 2020). github/semantic: Parsing, analyzing, and comparing source code across many languages, Github, [Online]. Available: https://github.com/github/semantic/tree/124b45d36d7f81e45a9aa3aef588fd5e132fb6fd (visited on 03/12/2020).

[44] (Mar. 2020). purescript/purescript: A strongly-typed language that compiles to JavaScript, Github, [Online]. Available: https://github.com/purescript/purescript/tree/183fc22549011804d973e01654e354b728f2bc70 (visited on 03/12/2020).

[45] (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).

[46] (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/0009512345fbd95fe1c745414fffed6c63ccd1aa (visited on 03/12/2020).

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

[48] (). Haskell : map, [Online]. Available: http://zvon.org/other/haskell/Outputprelude/map_f.html (visited on 04/10/2020).

[49] (Feb. 2020). fmt package - go.dev, [Online]. Available: https://pkg.go.dev/fmt@go1.14?tab=doc (visited on 04/12/2020).

[50] (). 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).

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

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

[53] (Mar. 2019). Foldr Foldl Foldl', [Online]. Available: https://wiki.haskell.org/index.php?title=Foldr_Foldl_Foldl%27&oldid=62842 (visited on 04/07/2020).

[54] (Mar. 2020). golangci/golangci-lint - Linters runner for Go, [Online]. Available: https://github.com/golangci/golangci-lint/commit/4958e50dfe8c95bebab5eaf360a3bc1fdc9574fe (visited on 04/18/2020).

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

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

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

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

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

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

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

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

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

[64] (Feb. 2020). builtin package - go.dev, [Online]. Available: https://pkg.go.dev/builtin@go1.14?tab=doc (visited on 03/13/2020).

[65] (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).

[66] (Apr. 2020). Comparing go1.14...bachelor-thesis - tommyknows/go, [Online]. Available: https://github.com/tommyknows/go/compare/go1.14...tommyknows:bachelor-thesis (visited on 04/13/2020).

[67] (Apr. 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#L3053 (visited on 04/13/2020).

[68] (Apr. 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#L3009 (visited on 04/13/2020).

[69] (Apr. 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#L3126 (visited on 04/13/2020).

[70] (Apr. 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#L3190 (visited on 04/23/2020).

[71] (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).

[72] (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).

[73] (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).

[74] (May 2020). funcheck/README.md at master - tommyknows/funcheck, [Online]. Available: https://github.com/tommyknows/funcheck/blob/master/prettyprint/README.md (visited on 05/10/2020).

[75] 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).

[76] (), Oracle, [Online]. Available: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html (visited on 05/16/2020).

[77] 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).

[78] 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).

[79] 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).

[80] (). Frequently Asked Questions, [Online]. Available: https://golang.org/doc/faq#Why_doesnt_Go_have_feature_X (visited on 05/21/2020).

[81] (2020). ruettram/bachelor, ZHAW, [Online]. Available: https://github.zhaw.ch/ruettram/bachelor (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. 13, 32

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. 24

DAG Directed Acyclic Graph. 13

Extended Backus-Naur Form An extended version of the ‘Backus-Naur Form, a notation technique to describe programming language syntax. 29

SSA Single Static Assignment, an intermediate representation between the AST and the compiled binary that simplifies and improves compiler optimisations. 14, 32

stdout Standard Output, the default output stream for programs. 48

Appendices

1 Example for Functional Options

Source Code 1: Functional Options for a simple Webserver

```
package webserver

type Server struct {
    Timeout time.Duration
    Port int
    ListenAddress string
}

type Option func(*Server)

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
    }
}

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

// usage examples from outside the package:
```

```
s := webserver.New() // uses all the default options
s := webserver.New(webserver.Timeout(time.Second), webserver.Port(8080))
```

2 Analysis of function occurrences in Haskell code

The results of the analysis have been aquired by running the following command from the root of the git repository[81]:

```
./work/common-list-functions/count-function.sh "map " " : " "fold"
↪  "filter " "reverse " "take " "drop " "maximum" "sum " "zip "
↪  "product " "minimum " "reduce "
```

3 Mutating variables in Go

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

```
1  package main

   import "fmt"

5  func main() {
     m := MyStruct{
       x: "struct",
       y: 42,
     }
10
     fmt.Println(m) // {struct 42}
     mutateNoPointer(m)
     fmt.Println(m) // {struct 42}
     mutatePointer(&m)
15   fmt.Println(m) // {changed 0}
   }

   type MyStruct struct {
```

```
  x string
  y int
}

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

4 Shadowing variables in Go

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

```
package main

import (
  "fmt"
)

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

```
    // this redeclares the variable x,
    // effectively shadowing it. This will
    // not change the parent blocks' variable.
    x := 4
    fmt.Println(x) // 4
  }
  fmt.Println(x) // 3
}
```

5 Workaround for the missing foldl' implementation in Go

Source Code 4: Working around the missing foldl implementation in Go

```
package main

import "fmt"

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
  // "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))
}

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

func mulLazy(x func() int, y *int) func() int {
```

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

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

6 Prettyprint implementation

Source Code 5: The original prettyprint implementation

```
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,
}

func run(pass *analysis.Pass) (interface{}, error) {
  for _, file := range pass.Files {
    ast.Inspect(file, func(n ast.Node) bool {
      switch as := n.(type) {
      case *ast.DeclStmt:
        fmt.Printf("Declaration %q: %v\n", render(pass.Fset, as),
        ↪  as.Pos())
        decl, ok := as.Decl.(*ast.GenDecl)
        if !ok {
          break
        }

        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(pass.Fset, val),
          ↪  val.Names[0].Pos())
        }
      case *ast.AssignStmt:
        type pos interface {
          Pos() token.Pos
        }

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

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

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

          // skip blank identifiers
          if ident.Name == "_" {
            fmt.Printf("\t\tBlank Identifier!\n")
            continue
          }

          if ident.Obj == nil {
            fmt.Printf("\t\tDecl is not in the same file!\n")
            continue
          }

          // 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(pass.Fset,
          ↪  ident.Obj.Decl), declPos)
        }
      }
      return true
    })
  }

  return nil, nil
}
```

```
80
// 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 {
85    panic(err)
  }
  return buf.String()
}
```

Source Code 6: The refactored, functional prettyprint implementation

```
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 {
      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()
}
```

7 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
```

7.1 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.

7.2 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.

7.3 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 ./...
```

8 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 7.

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 .
```

9 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 Komponenten)
  - (Berechnungen, Messwerte, Simulationsresultate)
  - (Stoffdaten)
  - (Fehlerrechnungen mit Messunsicherheiten)
  - (Grafische Darstellungen, Fotos)
  - (Datenträger mit weiteren Daten (z. B. Software-Komponenten) inkl. Verzeichnis der auf diesem Datenträger abgelegten Dateien)
  - (Softwarecode)