Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

Bachelor thesis in Informatik
Spring 2020

Functional Go

an easier introduction to functional programming

Authors	Ramon Rüttimann
Main supervisor	Prof. G. Burkert
Sub supervisor	Prof. K. Rege
Date	09.04.2020

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

When Javascript and Python started to take off around 2010[1], they also brought rise to a lot of concepts borrowed from functional programming. Since then, many new multi-paradigm languages have appeared and gotten more popular, as for example Go, Rust, Kotlin and Dart. Most of the languages mentioned support an imperative, object-oriented, as well as a functional programming style. Rust, being the 'most popular programming language'[2] for 4 years in a row (2016–2019), has been significantly influenced by functional programming languages[3] and borrows a lot of functional concepts in idiomatic Rust code.

Learning a functional programming language increases fluency with these concepts and teaches a different way to think and approach problems when programming. For those reasons, and many more, a lot of programmers advocate learning a functional programming language.

Though the exact definition of what a *purely* functional language consists of remains a controversy[4], the most popular, pure functional programming language seems to be Haskell[5].

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[6][7][8][9].

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[10]:

```
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

The goal is to solve the issue of the first steps in functional programming. 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:

- Make a multi-paradigm language support functional programming as much as needed. The criteria for this language are:
  - Easy, familiar syntax
  - Be statically typed, as this makes it easier to reason about a program
  - Have support for functional programming features like first class functions, currying and partial application
- Create a linter that checks code on its functional purity. For this, some rules will have to be curated to define what pure functional code is.

It s not the goal to create a production-ready functional language, so runtime and performance requirements can be ignored.

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[11]. With its strong syntactic similarity to C, it should be familiar to most programmers. Go is an extremely verbose language with almost no syntactic sugar. This makes it a perfect fit to grasp the concepts and trace the inner workings of functional programming.

There are, however, a few downsides of using Go:

- No polymorphism. Go 2 will likely have support for polymorphism, but at the time of writing, there is no implementation available.
- Missing implementations for common functions like 'map', 'filter', 'reduce' and more.
- No list implementation. Go has 'slices', which are 'views' on arrays, but no list datatype.

1.4.1 Go Slices

Go's Slices can be viewed as an abstraction over arrays, to mitigate some of the weaknesses of arrays 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.[12]*

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.

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[13].

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. This results in a number of helper functions[14] that currently do not exist in Go and would need to be implemented by the programmer. With no support for polymorphism, the programmer would need to implement a function 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[15]), 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.

To mitigate this point, the most common list-operations (in Go slice-operations) will be added to the compiler, so that the programmer can use these functions on every slice-type.

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)
}
```

[16]

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

A software design pattern that has gained popularity 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.

To give a quick example of how functional options look like and are used:

```go
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))
```

Dave Cheney's summary on functional options is thus:

> *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.*

[17]

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.

Functional Go?

In his talk 'Functional Go'[18], 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[19], the main issue was with the type system and the missing polymorphism.

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[20] 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!*

1.6 Work to be done

To conclude, the work that has to be done for this thesis is:

- Define which 'standard' list functions are most commonly used in functional programming
- Implement some of them into the compiler
  - at least one of these functions needs to have complete type-checking
  - demonstrate some usage examples for these functions
- Research 'rules' for pure functional programming
  - for example, this could be exact definitions of immutability and purity
- Add these rules to a checker. This could be an already available tool like 'go vet', or a newly developed, purpose-built utility

2 Methodology

– (Beschreibt die Grundüberlegungen der realisierten Lösung (Konstruktion/Entwurf) und die Realisierung als Simulation, als Prototyp oder als Software-Komponente)
– (Definiert Messgrössen, beschreibt Mess- oder Versuchsaufbau, beschreibt und dokumentiert Durchführung der Messungen/Versuche)
– (Experimente)
– (Lösungsweg)
– (Modell)
– (Tests und Validierung)
– (Theoretische Herleitung der Lösung)

2.1 Slice Helper Functions

2.1.1 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[21]:

- Shellcheck (koalaman/shellcheck[22]): A static analysis tool for shell scripts
- Pandoc (jgm/pandoc[23]): A universal markup converter
- Postgrest (PostgREST/postgrest[24]): REST API for any Postgres database
- Semantic (github/semantic[25]): Parsing, analyzing, and comparing source code across many languages
- Purescript (purescript/purescript[26]): A strongly-typed language that compiles to JavaScript

- Compiler (elm/compiler[27]): Compiler for Elm, a functional language for reliable webapps
- Haxl (facebook/haxl[28]): 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 2.1.

Table 2.1: Occurrences of list functions1

':' (cons)	2912
map	1241
fold	610
filter	262
reverse	154
take	104
drop	81
maximum	53
sum	44
zip	38
product	15
minimum	10
reduce	8

Based on this information, it has been decided to implement the cons, map and fold functions into the Go compiler[1]

[1]The implementation can be found at 'work/go' from the root of this git repository[29]. The work

2.1.2 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[30] that describes the function and it's usage
- Adding type-checking support in external packages for tools like Gopls[31]
- 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

2.1.3 Cons

Being the most commonly used function in Haskell code, Go needs an easier to use implementation for the cons operator than the currently existing `append([]<T>{elem}, src...)`. Keeping the naming scheme of go, the cons function will be named 'prepend', cohering with 'append'.

In regular Go code, a more efficient implementation compared to the example is:

Source Code 2.1: Prepend implementation in pseudo-Go code

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

is based upon Go version 1.14, commit ID '20a838ab94178c55bc4dc23ddc332fce8545a493'. All files referenced in this chapter are based from the root of the go repository, unless noted otherwise.

[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."[32]

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'.

This means that in the AST, the implementation should translate to:

Source Code 2.2: Prepend implementation in AST traversal

```
//   init {
//     dest := make([]<T>, 1, len(src)+1)
//     dest[0] = x
//     append(dest, src...)
//   }
//   dest
```

The type-checking for prepend is fairly simple. As seen in the Go implementation2.1, prepend accepts an element with type `T` and a slice with the element type `T`, and will return a slice of element type `T`. The AST node's type, once translated, will thus be `[]T`.

2.1.4 Fmap

Map is the second most-commonly used function in Haskell code. As 'map' is already a registered keyword within go's parser, the 'map' function will be called 'fmap' instead. This also improves readability and makes it easier to distinguish the 'map' type from the 'map' function.

A naive implementation of 'fmap', and it's improved version, is shown in 2.3

Source Code 2.3: Fmap implementation in pseudo-go code

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

```
}

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

Again, by using `make` to allocate the slice with it's full length at the beginning of the function, the calls to `append` and thus calls to grow the slice at runtime can be saved.

Similarily to prepend, the AST walk should translate the call to fmap to:

Source Code 2.4: Fmap AST translation

```
//   init {
//     dest := make([]out, len(src))
//     for i, e := range src {
//       dest[i] = f(e)
//     }
//   }
//   dest
```

Type-checking fmap should ensure that the given function's argument type is the same as the given slice's element type. Fmap returns a slice with the element type of the given function return type.

2.1.5 Fold

The third and last function that is implemented is fold. In Haskell, there are three distinct implementations of fold, 'foldr', 'foldl' and 'foldl'. 'foldr' allows, amongst other things, to transform infinite lists. 'foldl' and 'foldl' differ regarding their strictness properties. 'foldl' effectively reserves the given list, which means it returns the same result as 'foldr'

only if the given list is finite and the function commutative. It does however have better memory space properties than 'foldr'.

In Go, there are no infinite lists and no lazy evaluation. This means that a distinction between the different fold types in Go is not necessary, and the implementation will be based on the right fold, as:

> *'foldr' is not only the right fold, it is also most commonly the right fold to use,*

[33]

A pseudo-go implementation of 'fold' looks like this:

Source Code 2.5: Fold implementation in pseudo-go code

```
func fold(f func(Type, Type1) Type1, acc Type1, src []Type) Type1 {
    for i := len(src) - 1; i >= 0; i-- {
        acc = f(src[i], acc)
    }
    return acc
}
```

This walks the given slice backwards, calling the function and updating the acculumator with the new value.

The AST walk translates fold to:

Source Code 2.6: Fold AST translation

```
//   init {
//     for i := len(s) - 1; i >= 0; i-- {
//       acc = f(s[i], acc)
//     }
//   }
//   acc
```

The type-check for fold will need to ensure that the given function's first argument type corresponds to the slice's element type, that the second argument is the same as the function's return type and that the accumulator is of the same type as well.

2.2 Functional Check

3 Implementation

3.1 Slice Helper Functions

The new builtin functions prepend, fmap and fold have to be registered and implemented in different places within go's source code.

The go source code that is relevant for this thesis can be classified into two different types. One group is the 'public' implementation of these builtins, and one is the 'private' implementation.

The 'private' implementation is everything that is located within the *src/cmd/compile/internal* package[32]. It can only be used by the package in *src/cmd/compile*, which contains the implentation of the compiler itself. When calling `go build .`, the compiler is invoked indirectly. To directly invoke the compiler, `go tool compile` can be used. The compile tool gets compiled from the main package located in *src/cmd/compile/main.go*, which in turn uses the internal package.

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.

3.1.1 Implementing Prepend

Adding the GoDoc

In Go, documentation is generated directly from comments within the source code[30]. This also applies to builtin functions in the compiler, which have a function stub to document their behaviour[34], but no implementation, as that is done in the compiler[35].

The prepend function header is modelled after the append declaration:

Source Code 3.1: Godoc for builtin functions (*src/builtin/builtin.go*)

```
func append(slice []Type, elems ...Type) []Type
func prepend(elem Type, slice []Type) []Type
```

The file containing the documentation is '*src/builtin/builtin.go*'. Adding a comment above the 'prepend' function header concludes the documentation part of the implementation.

Adding prepend to the public packages

The second step is to add prepend to the '*src/go/types*' package.

> *Package types declares the data types and implements the algorithms for type-checking of Go packages*

[36]

Though it will not have a direct impact on the compiler, adding the prepend function to the *types* package allows external tools like Gopls to type-check '.go' files and support the programmer at writing code.

To allow type-checking prepend, the function has to be registered as a builtin. This is done in '*src/go/types/universe.go*':

Source Code 3.2: public type-checking for prepend builtin (*src/go/types/universe.go*)

```
107 // ...
    // A builtinId is the id of a builtin function.
    type builtinId int
110
    const (
      // universe scope
      _Append builtinId = iota
      _Prepend
115   _Fmap
      _Fold
      // ...
```

universe.go also contains a section for `predeclaredFuncs`, which define the number of arguments, type and name for builtin functions:

Source Code 3.3: list of predeclared functions (*src/go/types/universe.go*)

```
143 // ...
var predeclaredFuncs = [...]struct {
145   name     string
      nargs    int
      variadic bool
      kind     exprKind
}{
150   _Append:  {"append", 1, true, expression},
      _Prepend: {"prepend", 2, false, expression},
      _Fmap:    {"fmap", 2, false, expression},
      _Fold:    {"fold", 3, false, expression},
      _Cap:     {"cap", 1, false, expression},
155   // ...
```

Prepend takes two arguments and is not a variadic function. The expression kind can either be a 'conversion', 'expression' or a 'statement'. As it returns a value, prepend is an expression, similar to append.

The existing unit tests within the 'types' package ensure that every builtin function also has at least one test. This makes it mandatory to add a test for 'prepend' too:

Source Code 3.4: builtin functions tests (*src/go/types/builtins_test.go*)

```
27 // ...
{"prepend", `var s []int; _ = prepend(0, s)`, `func(int, []int) []int`},
{"prepend", `type T int; var s []T; var n T; _ = prepend(n, s)`,
↪  `func(p.T, []p.T) []p.T`},
30 {"prepend", `var s []int; _ = (prepend)(0, s)`, `func(int, []int)
↪  []int`},
// ...
```

The type-checking itself is straightforward. The implementation starts by checking that the second argument is a slice, and then extracts the slice's element type and checks that the first argument is of the same type.

Source Code 3.5: type checking prepend (*src/go/types/builtins.go*)

```
// ...
case _Prepend:
  // prepend(x T, s S) S, where T is the element type of S
  // spec: prepend is like append, but adds to the beginning instead of
  ↪  the end of the slice.
  // The values x are passed to a parameter of type T where T is the
  ↪  element type
  // of S and the respective parameter passing rules apply."

  // the second argument is the slice, so we start by getting that type.
  arg(x, 1)
  S := x.typ
  var T Type
  if s, _ := S.Underlying().(*Slice); s != nil {
    T = s.elem
  } else {
    check.invalidArg(x.pos(), "%s is not a slice", x)
    return
  }

  // save the already evaulated argument
  arg1 := *x
  // reset to the first argument
  arg(x, 0)

  // check general case by creating custom signature
  sig := makeSig(S, T, S)
  check.arguments(x, call, sig, func(x *operand, i int) {
    // only evaluate arguments that have not been evaluated before
    if i == 1 {
      *x = arg1
      return
    }
    arg(x, i)
  }, nargs)
  // ok to continue even if check.arguments reported errors

  x.mode = value
  x.typ = S
  if check.Types != nil {
```

```
    check.recordBuiltinType(call.Fun, sig)
  }
// ...
```

This concludes the type-checking for external tools and makes 'gopls' return errors if the wrong types are used[1]. 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"}))
}
```

Gopls then reports the type-checking error:

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

Adding prepend to the private compiler package

The compiler is implemented at *src/cmd/compile/internal/gc*[2], so all mentioned files are located within this directory.

First, prepend needs to be registered as a builtin to parse prepend into the AST (technically, the syntax tree is a syntax DAG[37], but this is an implementation detail):

Source Code 3.6: registering prepend in the compiler (*universe.go*)

[1] For this to work, 'gopls' has to be compiled against the modified version of the go source. This means pointing the go toolchain to the correct directory by setting the value of 'GOROOT' with `go env -w GOROOT=<path>`.

[2] 'gc' within the path stands for 'go compiler', and not 'garbage collection'.

```
44 // ...
45 var builtinFuncs = [...]struct {
     name string
     op   Op
   }{
     {"append", OAPPEND},
50   {"prepend", OPREPEND},
     {"fmap", OFMAP},
     {"fold", OFOLD},
     {"cap", OCAP},
     // ...
```

'OPREPEND' is an operation that has to be defined in *syntax.go*:

Source Code 3.7: OPREPEND node definition

```
609 // ...
610 type Op uint8

    // Node ops.
    const (
      OXXX Op = iota
615   // ...
```

```
632   // ...
      OAPPEND    // append(List); after walk, Left may contain elem type
      ↪  descriptor
      OPREPEND   // prepend(Left, Right)
635   OFMAP      // fmap(Left, Right); a typical map function from FP, left
      ↪  is the func, right the slice
      OFOLD      // fold(List)
      OBYTES2STR // Type(Left) (Type is string, Left is a []byte)
      // ...
```

This also shows that an 'OPREPEND' node expects its arguments to be in 'node.Left' and 'node.Right', and not, like 'OAPPEND', in 'node.List'. This is achieved in *type-*

check.go. Before typechecking, a function's arguments are always defined in 'node.List'. In the implementation for 'OPREPEND', the arguments are typechecked and parsed into 'node.Left' and 'node.Right' by the statements:

```
// typecheck the arguments, expand function arguments etc.
typecheckargs(n)
// twoarg ensures there's exactly two arguments and adds
// them to n.Left and n.Right
if !twoarg(n) {
    // ...
}
```

After 'node.Left' and 'node.Right' are populated, 'node.Right'ś type is checked to be a slice, and 'node.Left' is of the same type as the slice elements. The full typecheck is added as case within *typecheck.go*:

Source Code 3.8: typechecking prepend (*typecheck.go*)

```
1578 // ...
     case OPREPEND:
1580   ok |= ctxExpr
       typecheckargs(n)
       if !twoarg(n) {
         n.Type = nil
         return n
1585   }

       t := n.Right.Type
       if t == nil {
         n.Type = nil
1590     return n
       }

       n.Type = t
       if !t.IsSlice() {
1595     if Isconst(n.Right, CTNIL) {
           yyerror("second argument to prepend must be typed slice; have
           ↪  untyped nil")
           n.Type = nil
           return n
```

```
    }

    yyerror("second argument to prepend must be slice; have %L", t)
    n.Type = nil
    return n
  }

  if n.Left.Type == nil {
    yyerror("first Argument has no type")
    n.Type = nil
    return n
  }
  n.Left = assignconv(n.Left, t.Elem(), "prepend")
  checkwidth(n.Left.Type)
// ...
```

With this, type-checking prepend in the AST is implemented. The next step in the Go compiler is escape analysis. This is similar to the 'OAPPEND' node, though 'OPREPEND' always has only two arguments:

Source Code 3.9: escape analysis for prepend (*escape.go*)

```
// ...
case OPREPEND:
  // imitate the behaviour from OAPPEND
  paramKs[0] = e.heapHole()

  if types.Haspointers(call.Right.Type.Elem()) {
    paramKs[1] = e.teeHole(paramKs[1], e.heapHole().deref(call,
    ↪  "prepended slice"))
  }
// ...
```

The next phase within the compiler are AST transformations to lower the AST to a more easily compilable form. The first call is made to 'order'. 'order' reorders expressions and enforces the evaluation order. For prepend, this is done by adding another 'case' expression within `func (o *Order) expr(n, lhs *Node) *Node`. Again, this can be implemented similarly to append:

Source Code 3.10: ordering the prepend arguments (*order.go*)

```
1181 // ...
     case OPREPEND:
       n.Left = o.expr(n.Left, nil)
       n.Right = o.expr(n.Right, nil)
1185
       if lhs == nil || lhs.Op != ONAME && !samesafeexpr(lhs, n.Left) {
         n = o.copyExpr(n, n.Type, false)
       }
     // ...
```

The compiler now calls 'walk' to do more AST transformations, for example replacing nodes like 'OAPPEND' with the actual implementation of the algorithm, in AST form. This is where the logic for 'OPREPEND' needs to be added too. To recapitulate, the general algorithm for 'prepend' is:

```
dest := make([]<T>, 1, len(src)+1)
dest[0] = elem
return append(dest, src...)
```

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

Source Code 3.11: implementation of prepend within the AST (*walk.go*)

```
2996 // ...
     // walkprepend rewrites the builtin prepend(x, src) to
     //
     //   init {
3000 //     dest := make([]<T>, 1, len(src)+1)
     //     dest[0] = x
     //     append(dest, src...)
     //   }
     //   dest
3005 //
     func walkprepend(n *Node, init *Nodes) *Node {
       tail := temp(n.Right.Type)
```

```
  var l []*Node
  l = append(l, nod(OAS, tail, n.Right))

  // length is always one, the element that is prepended
  makeLen := nodintconst(1)                                    // len = 1
  makeCap := nod(OADD, nodintconst(1), nod(OLEN, tail, nil)) // cap =
  ↪  len(tail) + 1
  // get the type of the tail
  makeType := nod(OTYPE, nil, nil)
  makeType.Type = tail.Type

  makeDest := nod(OMAKE, nil, nil)
  makeDest.List = asNodes([]*Node{makeType, makeLen, makeCap}) //
  ↪  make([]<T>, 1, len(tail) + 1)

  // create the destination slice
  ndst := temp(tail.Type)

  l = append(l, nod(OAS, ndst, makeDest))                          //
  ↪  ndst = make([]T, 1, len(tail)+ 1)
  l = append(l, nod(OAS, nod(OINDEX, ndst, nodintconst(0)), n.Left)) //
  ↪  ndst[0] = x

  // type check and walk everything
  typecheckslice(l, ctxStmt)
  walkstmtlist(l)
  init.Append(l...)

  a := nod(OAPPEND, nil, nil)
  a.List = asNodes([]*Node{ndst, tail})
  return appendslice(a, init) // append(ndst, tail)
}

// ...
```

This ‘walkprepend’ function is called from ‘walkexpr’. Within ‘walkepr’, it is called only if the parent node is of the operation ‘OAS’ or ‘OASOP’, as prepend is an expression that always returns a value, meaning it must be assigned back.

This concludes the implementation of prepend. Some changes may have been omitted and can be found by comparing the changes between the git tag ‘go1.14’[38] and ‘ba-added-prepend’[39].

4 Application

5 Experiments and Results

(Zusammenfassung der Resultate)

6 Discussion

– Bespricht die erzielten Ergebnisse bezüglich ihrer Erwartbarkeit, Aussagekraft und Relevanz
– Interpretation und Validierung der Resultate
– Rückblick auf Aufgabenstellung, erreicht bzw. nicht erreicht
– Legt dar, wie an die Resultate (konkret vom Industriepartner oder weiteren Forschungsarbeiten; allgemein) angeschlossen werden kann; legt dar, welche Chancen die Resultate bieten

Bibliography

[1] David Robinson. (Sep. 6, 2017). The incredible growth of Python, [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/?_ga=2.199625454.1908037254.1532442133-221121599.1532442133 (visited on 02/29/2020).

[2] 'Stack Overflow Developer Survey 2019,' Stack Overflow, Tech. Rep., 2019. [Online]. Available: https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted.

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

[5] (Feb. 7, 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).

[6] (Jun. 29, 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).

[7] I. Connolly. (Feb. 7, 2014). A lazy evaluation, [Online]. Available: http://connolly.io/posts/lazy-evaluation/ (visited on 03/04/2020).

[8] (Oct. 23, 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).

[9] (Sep. 12, 2007). The Haskell learning curve, [Online]. Available: https://wiki.haskell.org/index.php?title=Humor/LearningCurve&oldid=15543 (visited on 03/04/2020).

[10] (Feb. 29, 2020). Introduction - Haskell Wiki, [Online]. Available: https://wiki.haskell.org/index.php?title=Introduction&oldid=63206 (visited on 03/01/2020).

[11] Jason 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).

[12] Andrew Gerrand. (Jan. 5, 2011). Go Slices: usage and internals, [Online]. Available: https://blog.golang.org/go-slices-usage-and-internals (visited on 02/29/2020).

[13] (Jan. 15, 2020). SliceTricks, [Online]. Available: https://github.com/golang/go/wiki/SliceTricks/a87dff31b0d8ae8f26902b0d2dbe7e4e77c7e6e3 (visited on 03/04/2020).

[14] (Nov. 15, 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).

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

[16] (Feb. 25, 2020). http package - go.dev, [Online]. Available: https://pkg.go.dev/net/http@go1.14?tab=doc#example-HandleFunc (visited on 03/06/2020).

[17] D. Cheney. (Oct. 17, 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).

[18] Francesc Campoy Flores. (Dec. 11, 2015). Functional Go? Youtube, [Online]. Available: https://www.youtube.com/watch?v=ouyHp2nJl0I.

[19] I. L. Taylor. (Nov. 7, 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).

[20] (Nov. 7, 2019). go-functional/core, Github, [Online]. Available: https://github.com/go-functional/core/tree/700b20aec09da808a67cc29ae2c54ad64f842851 (visited on 03/06/2020).

[21] (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).

[22] (Mar. 8, 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).

[23] (Mar. 10, 2020). jgm/pandoc: Universal Markup Converter, Github, [Online]. Available: https://github.com/jgm/pandoc/tree/91f2bcfe73fa3a489654ee74bca02e24423dc5c0 (visited on 03/12/2020).

[24] (Mar. 7, 2020). PostgREST/postgrest: REST API for any Postgres database, Github, [Online]. Available: https://github.com/PostgREST/postgrest/tree/dea57bd1bec9ba5c4e438de80b2017eee4a30a40 (visited on 03/12/2020).

[25] (Mar. 11, 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).

[26] (Mar. 11, 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).

[27] (Mar. 11, 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).

[28] (Mar. 3, 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).

[29] (2020). ruettram/bachelor, ZHAW, [Online]. Available: https://github.zhaw.ch/ruettram/bachelor (visited on 03/13/2020).

[30] A. Gerrand. (Mar. 31, 2011). Godoc: documenting Go code, [Online]. Available: https://blog.golang.org/godoc-documenting-go-code (visited on 03/13/2020).

[31] (Dec. 17, 2019). gopls documentation, [Online]. Available: https://github.com/golang/tools/blob/0b43622770f0bce9eb6c5d0539003ebbc68b9c70/gopls/README.md (visited on 03/20/2020).

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

[33] (Mar. 29, 2019). Foldr Foldl Foldl', [Online]. Available: https://wiki.haskell.org/index.php?title=Foldr_Foldl_Foldl%27&oldid=62842 (visited on 04/07/2020).

[34] (Feb. 25, 2020). builtin package - go.dev, [Online]. Available: https://pkg.go.dev/builtin@go1.14?tab=doc (visited on 03/13/2020).

[35] (Apr. 5, 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).

[36] (Oct. 17, 2019). types package - go.dev, [Online]. Available: https://pkg.go.dev/github.com/golang/go@v0.0.0-20191017215157-9e6d3ca2794c/src/go/types?tab=doc (visited on 03/20/2020).

[37] (Nov. 12, 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).

[38] (Feb. 25, 2020). tommyknows/go at ba-go-1-14, [Online]. Available: `https://github.com/tommyknows/go/tree/go1.14` (visited on 03/21/2020).

[39] (Mar. 21, 2020). tommyknows/go at ba-added-prepend, [Online]. Available: `https://github.com/tommyknows/go/tree/ba-added-prepend` (visited on 03/21/2020).

List of source codes

List of Figures

List of Tables

Glossary

AST Abstract Syntax Tree. 13

SSA Single Static Assignment, an intermediate representation between the AST and the compiled binary that simplifies and improves compiler optimisations. 13

Appendices

1 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[29]:

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

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)