COMPARE FUNCTIONAL PROGRAMMING IN GO WITH PYTHON

JONAS BAUER*

* *Rosenheim, Technical University of Applied Sciences, Subject: Concepts of Programming Languages*

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

The last years have seen a remarkable rise in the popularity of functional programming, marking its transition from a niche approach to an integral part of contemporary programming practice. This surge has been underlined by the emergence of new languages such as Scala or Clojure, which are specifically tailored for the Java Virtual Machine. In addition, the adoption of functional programming elements is widespread in today's programming landscape, transcending language boundaries. Understanding and incorporating functional programming principles has become increasingly important, reflecting their relevance and the shift towards a more functional style as an industry standard. [1]
In response to these developments, this paper seeks to explore the theoretical foundations and practical implementations of functional programming. The principal objective is to investigate how functional programming manifests in modern languages, with a focused examination of Golang (or for short: Go) and Python. For the purpose of the investigation, the game of checkers [2] was implemented using the minimax algorithm [3] for the opposing player. Some of the code snippets shown in this paper come from this implementation. Consequently, our examination will commence with an exploration of the foundational principles of functional programming before delving into the specific implementations in the two languages. Subsequently, we will conduct a comparative analysis of the two implementations, leading to a conclusion regarding the instantiation and distinctions of the paradigm in each language.

2 FOUNDATIONS OF FUNCTIONAL PROGRAMMING

In functional programming, functions play a central role, and the approach is to break down a problem into a set of functions. Ideally, functions within this paradigm take only inputs and produce only outputs, with no internal state that could affect the output for a given input. Functional programming languages are categorised according to their adherence to purity (see Chapter 2.3). Languages in which internal state and side effects are impossible or at least very difficult to achieve are called 'pure', while those that allow such features are called 'impure'. In the object-oriented paradigm, on the other hand, it is essential to manipulate collections of objects that have an internal state and support methods for querying or modifying that state. Functional programming can be thought of as the opposite of

object-oriented programming. While objects are little capsules containing some internal state, along with collections of methods that modify that state, and programs consist of making the right set of state changes, functional programming tries to avoid state changes as much as possible, and works with data flowing between functions. [4] The following will introduce the basics of functional programming before diving into the concrete implementation in Python and Go.

2.1 Functions as First Class Citizens

A first-class citizen in a programming language is an entity such as an object, primitive or function that works with all common language operations. These operations include assigning it to a variable, passing it to a function, returning it from a function, or storing it in another data type such as a map. In a pure object-oriented programming (OOP) language, such first-class citizens are objects, for example. They can be passed between functions, they are often returned as the result of a function and assigned to a variable. So when we say that functions are First Class Citizens, we can think of them as being treated like objects. In the OOP paradigm, functions also exist. They are associated with an object and are used to describe the functionality of an object. They can change the state of an object to which they are bound. Therefore, in contrast to the functional programming paradigm, they are so-called secondary citizens in OOP. [5, Chapter1]

2.2 Higher-Order Functions

The concept of Higher-Order Functions (HOF) is based on treating functions as first-class citizens. A HOF is basically any function that that either takes a function as an input or returns a function as an output.
With Higher-Order Functions it is possible to apply techniques such as function closure or function currying.

FUNCTION CLOSURES Function closures are related to how variable scoping works in a programming language. So for most programming languages, this means that variables defined in a "are available at a lower location", but variables at the lower location are not available at the higher location. A closure is then defined by any inner function that uses a variable introduced in the outer function to do its work. [5, Chapter3]
Another way of thinking about closures is to compare it to the object-oriented approach. So we can say that a class is "data with operations", while a closure is "operations with data attached". [6, Chapter2]

```
// adder returns a closure that adds x to its argument.
func adder(x int) func(int) int {
   // The returned function (closure) has access to the
       variable squared.
   squared := x*x
   return func(y int) int {
      x += y
      return x
   }
}
func main() {
   // Creating a closure with an initial value of 10.
   adderWithInitialValueOf10 := adder(10)
   result := adderWithInitialValueOf10(5)
   }
```

Figure 1: Example of Function Closure

The figure 1 shows an example of a function closure. The adder in this case is a Higher-Order function because it returns a function (the closure) as output. The inner function is an "anonymous function", that means it has not specific name, which has its own arguments. It also has access to the parameter squared, which is defined in the outer function. In the main function we declare a closure by calling the adder function with a value of 10. So calling the *adderWithInitialValueOf10(5)* would result in the calculation of $10 * 10 + 5 = 105$. Partial application is a concept closely related to closure. It means that a function is only partially applied. So a function has N arguments. With partial application we fix a subset of the arguments, while the other arguments remain flexible. So the difference here is that partial application is used to create functions with fewer arguments, while function closure is a more general concept that retains access to variables from the outer scope even after the scope has finished executing. It provides a way of encapsulating and persisting data. [5, Chapter3]

FUNCTION CURRYING Function Currying is often mistaken for partial application. It is a very similiar concept. The main difference is that partial functions are fixing a subset of parameters to return a function with less parameters whereas function currying can be seen as transforming a function with multiple parameters into a sequence of functions, each taking a single parameter. In Haskell for example each function is transformed into a curried function by the compiler. [5, Chapter3]

TYPES OF HIGHER-ORDER FUNCTIONS There are many Higher-Order functions. However, many of them can be grouped into the following three categories.

- predicate-based functions: A predicate is a statement that can be either true or false. Predicate-based functions are Higher-Order functions and typically use such predicates to to determine which items in a given collection meet certain criteria. This can be used, for example, to filter out items based on a condition, or to check if all items meet a particular condition.
- map/transformation functions: These are functions that apply a transformation function to each element in a container, changing the element and possibly even the data type. For example, we want to multiply each element by 2.
- reduce functions: These are functions that apply an operation to a container of items and get a single value from them. A simple example would be the sum of all elements in an array. [5, Chapter6]

2.3 Pure Functions

Pure Functions are completely deterministic. That means the output of a function is entirely determined by its input. There is no randomness or uncertainty. In other words, the function returns the same output when the same input is provided.

Pure Functions also produces no side effects. That means they do not alter external state or rely on changes in the environment.

To illustrate what it means, a short example of an impure function in Golang.

```
var globalVariable int
func rollDice() int {
   randomNumber := rand.Intn(6)
   globalVariable = randomNumber
   return randomNumber
}
```

Figure 2: Example of an impure function

The function in figure 2 is not deterministic. If you roll the dice twice, it is very likely that you will get different results. So the result of the function is dependent on randomness. It also has side effects. When the function is called, it is likely that the variable *globalVariable* will change its state.

Other properties of pure functions are idempotence and statelessness. Idempotence means that calling the pure function multiple times does not change the output at all. In other words, multiple identical calls have the same effect as a single one. Statelessness means that a pure function does not depend on any state of the system. Neither the

input nor the output should change the state of the system. [5, Chapter4]

BENEFITS Using pure functions brings a lot of Benefits. For example, pure functions are easier to test because they are idempotent. We always get the same result for our function, no matter how many times we call it. It would be hard for us to write tests for functions that return different outputs for the same inputs. Another property that improves testability is statelessness. If our functions are stateful and we rely on the state of the system in some way, we should also guarantee that our test environment replicates the production state.
Pure functions also allow easier debugging and reasoning, because their behaviour is determined solely by their inputs, and they have no hidden dependencies on external state. [5, Chapter 4]

2.4 Immutability

Immutability ensures that once an instance of a data structure is created, its state remains constant throughout its lifetime. If it is necessary to modify this instance, a new instance is created with the desired changes, rather than modifying the existing one. There are several advantages to this approach. First, the fact that our object remains unchanged allows us to pass it to other functions with confidence. No matter what happens within those functions, the original object remains unchanged. Another important benefit is that it makes it easier to write correct concurrent code. Because the state of an immutable object cannot be changed by any function that calls it, we can parallelise execution more easily and call multiple functions on the same object. In contrast, using mutable objects in a concurrent environment can lead to problems such as data races, inconsistencies and bugs. In addition, using mutable objects can introduce unwanted side effects. Changing an existing object changes the state of the system, which can have unintended consequences.[5, Chapter 5]

2.5 Recursion

Recursion is a programming concept in which a function calls itself during execution. A recursive function consists of two main components: the recursive part, which defines when the function calls itself, and the base case, which acts as a stop condition to prevent infinite loops.

```
func Fact(input int) int {
   if input == 0 {return 1}
   return input * Fact(input-1)
}
```

Figure 3: Example of a recursive function

The figure 3 shows how to calculate the factorial of a number using a recursive function. The factorial of 0 is 1. This is our base case. When the function is called with $input = 0$, we return the value 1. In all other cases, we multiply the input by the value of the factorial function with $input = input - 1$. What happens is that each time *Fact(input int)* is called, a function call is pushed onto our stack. Each recursive call works towards the base case, and once the base case is reached, the results are propagated back up the stack from top to bottom. Any lower level of the stack can use the result from what come above.

ADVANTAGES Recursion has advantages over iterative solutions. It is more aligned with functional programming principles. Although iterative solutions can be rewritten as recursive ones and vice versa, iterative solutions have to maintain more state than recursive ones. If our primary goal is to improve the readability and maintainability of code, then recursion can make it easier to write and understand programs in certain scenarios. For example, when dealing with algorithms based on recursive data structures such as graphs and trees. Recursion provides an effective approach to dealing with such cases. Expressing this algorithm in an iterative fashion would likely result in reduced readability and maintainability.

DISADVANTAGES In general, with recursion we have increased time and space requirements. Calling a function multiple times creates multiple stack frames. Each stack frame contains a copy of the data of the frame below. All these frames are alive at the same time. A recursive call stack does not pop the stack until the last recursive call has finished. The stack also has a finite amount of space. If all the data from the current iteration has to be copied over and passed to the new function, it is possible to get a stack overflow. A stack overflow means that there is no more room on the stack and the program will eventually stop. Tail Call Optimisation (TCO) is a strategy that aims to avoid allocating a new stack frame for recursive function calls. By using this technique, we can facilitate infinite recursion. If TCO is supported in a prog2ramming language, we can convert our function into a tail-call recursive function and the compiler will automatically detect and optimise it. [5, Chapter7]

2.6 Lazy Evaluation

When a programming language supports lazy evaluation of a function call, it means that the function is only executed at the moment the result is needed instead of ahead of time. In contrast, eager evaluation (or strict evaluation) involves computing the entire result for each function at the time of the function call. [5, Chapter8]

2.7 Monads

A monad is a design pattern; it's a special way of organising and managing data. What makes a monad special is its ability to perform two crucial tasks [5, Chapter5]:

- It can take a regular thing (of type T, such as a string) and put it in a special container called Monad[T].
- It can also take a function designed for regular things and adapt it to operate on elements inside the special container Monad[T].

Example: Let's consider the Maybe or Optional monad, a widely used type. This monad is useful when dealing with something that may or may not exist. Imagine a box (Monad[T]) that can contain a real object or be empty. The box has two primary functions: It can take a regular thing and encapsulate it, effectively wrapping a value of type T into the monadic type Maybe[T]. It can take a function designed for regular things and modify it to work seamlessly on elements inside the box, essentially making the function compatible with Maybe[T] types. For example, if you want to add 1 to a number, but that number might not exist, the monad type, in this case the Maybe monad, facilitates the operation without having to worry about the existence of the number.

Monads are not the focus of this essay due to its complexity. So, we will further not consider any implementation in Python or Golang.

3 FUNCTIONAL PROGRAMMING IN GOLANG

Go is invented by Google as a general-purpose language with systems programming in mind. It is strongly typed and garbage-collected and has explicit support for concurrent programming. [7] Go is a multi-paradigm programming language, supporting object-oriented, functional and imperative approaches to problem-solving. While it embraces functional programming concepts, Go is considered impure due to its allowance of side effects and mutable state. It is statically typed, meaning that variable types are determined at compile-time.

[8] In the following, we will explore how Go implements the foundational concepts, mentioned in chapter 2 of functional programming. Concrete examples and code snippets will be provided to illustrate these implementations.

FUNCTIONS AS FIRST CLASS CITIZENS In Go, functions are treated as first-class citizens, so we can work with them as we would with any other data type. This means that all the common operations mentioned earlier in the 2.1 section, such as assigning to a variable, passing to a function, returning from a function, or storing in another data type, also work on functions. In addition, Go has type aliases, allowing you to give a certain type of function a name for increased clarity and abstraction. Figure 4 illustrates an example of using type aliases with functions. [5, Chapter2]

```
//Example of a type alias
type PieceFunction func(row, column int) Piece
```

Figure 4: An example of a type alias

HIGHER-ORDER FUNCTIONS Because Go treats functions as first-class citizens, it has the ability to define higher-order functions, allowing functions to take other functions as input or return functions as values. Techniques such as function closure, partial application, or function currying, as discussed in the chapter 2.2, can be used. Typical higher-order functions such as map, reduce, or predicate-based functions (see 2.2) are not part of the standard Go implementation. Prior to the introduction of generics in 2021, developers had to implement common higher-order functions such as map, reduce, and filter specifically for their data structures. This involved either custom implementations or the use of third-party libraries such as "pie" [9], which generated code tailored to the specific data structure. With the advent of generics, functional programming libraries became popular, making functional programming in Go more convenient [5, Chapter11]. In particular, a library called "lodash" has emerged for Go [10]. As well as third-party libraries, the standard library is also getting more support for higher-order functions, as the latest language update shows. In Go 1.21, a notable addition to the standard library is the extension for slices on any element type, providing enhanced support for higher-order functions. [11]

IMMUTABILITY To ensure immutability in Go, especially with structs, a full understanding of pointers, pass-by-value and pass-by-reference is essential. The use of pointers allows direct modification via the pass-by-reference approach, where the function operates on the mem-

ory address where the struct is stored, resulting in changes to the original struct in memory.

To maintain immutability, a preferable approach is to use pass-by-value. In this scenario, the original struct remains unchanged because the function operates on a copy of the struct. Modifications are applied to the copy and a new struct is returned, preserving the immutability of the original struct. Figure 5 shows an example of both cases. The first function uses the pass by value approach. The second uses pass by reference.

```
type GameBoard struct {
   GameBoard [][]Piece
   CurrPlayer string
}
//Works on a copy
func (gameboard GameBoard)
    generateBoard(get_piece_with_team_function PieceFunction)
    [][]Piece {
  board := lo.Map(gameboard.GameBoard, func(item []Piece, row
     int) []Piece {
      return generateRow(get_piece_with_team_function, row)
})
return board}
 //Works on the original Gameboard because of the Pointer usage
 func (gameboard *GameBoard)
    generateBoardWithPointer(get_piece_with_team_function
    PieceFunction) [][]Piece {
  board := lo.Map(gameboard.GameBoard, func(item []Piece, row
     int) []Piece {
      return generateRow(get_piece_with_team_function, row)
})
return board}
```

Figure 5: Difference between Pass by Value and Pass By Reference in Go

There are other mutable data structures in Go, such as maps or slices. It is important to note that maps always behave like pass by reference. It does not matter how we pass the map to the function, whereas slices work as expected. [5, Chapter5]

RECURSION As in many other impure functional languages, recursive solutions in Go can have increased time and space requirements, as discussed in the chapter 2.5. It's worth noting that although Go has a relatively large default stack space (1GB on 64-bit systems and 250MB on 32-bit systems), it lacks tail call optimisation, making stack overflows possible. Experimental evidence suggests that recursive solutions in Go tend to be slower than their iterative counterparts. For example, computing the factorial of 10 takes about four times as long

with a recursive implementation as with an iterative one. [5, Chapter7]

LAZY EVALUATION Go internally does not use lazy evaluation out of the box. This means that it will not automatically translate our code into functions called with lazy evaluation. However, it is possible to force Go to do this by using higher-order functions. [5, Chapter8]

4 FUNCTIONAL PROGRAMMING IN PYTHON

Python is a multi-paradigm language. Multi-paradigm means that you can write programs in Python that are largely procedural, object-oriented, functional, or a mixture of these approaches. [4] Unlike statically typed languages like Go (see 3) where variables types are determined at compile time, in Python the type of a variable is determined at runtime based on the value assigned to it. [12, Chapter 2.2] From a functional programming perspective, Python is classified as an impure functional language, which acknowledges its support for functional programming concepts while allowing side effects and mutable state [4] This section shows how the basics from the chapter 2 are implemented and what more sophisticated methods are available in Python.

FUNCTIONS AS FIRST CLASS OBJECTS In Python, functions are objects that are typically created using the def statement and can be manipulated by other functions. In addition, Python provides flexibility in creating functions, allowing them to be created as callable objects or assigned to variables using lambda expressions. Because functions are objects in Python, they can be manipulated and used in the same way as other objects in the language. For example, a function can be assigned to a variable, functions can be passed as arguments to other functions, and it is possible to get a function as a return value from another function call. This feature qualifies functions as so-called first-class objects. [13, Chapter 2]

HIGHER-ORDER FUNCTIONS Because functions are treated as first-class objects, it is possible to pass functions to functions or return functions from a function, Python has full support for higher-order functions. Techniques such as function currying, partial application and function closure exist in Python. Predicate based functions such as filters, map/transform functions in general, reduce functions and many more are part of the standard library in Python. They can be used right out of the box without having to install a third-party library. [6, Chapter4] Nonetheless, the majority of Python programmers ap-

pear to favor comprehensions over map, reduce, and filter. Another feature that Python has is decorators. Decorator design techniques are very common in Python, probably the most common use case of higher-order functions. It is a syntax sugar for a higher-order function which input and output is a function. When a decorator is applied to a function, it can perform actions before, after, or around the execution of the function, effectively allowing you to separate concerns without modifying the function's source code.

```
def message_decorator(func):
   def wrapper():
      print(f"Before")
      func()
      print(f"After")
   return wrapper

@message_decorator
def function_decorator():
   print("Example function")

function_without_decorator =
    message_decorator(print("example function"))
```

Figure 6: Difference between Pass by Value and Pass By Reference in Go

Figure 6 shows an example of a decorator. By calling *function_decorator* it will print "Before", "Example function", "After". It is worth mentioning that the example *function_without_decorator* and *function_decorator* are equivalent because of the decorator *@message_decorator()* used in *function_decorator*.

IMMUTABILITY Python has both mutable and immutable data types. In functional programming we rely on immutable data types, as mentioned in the chapter 2.4. While lists, dictionaries (hashmap implementation in Python) and sets are mutable data structures, numbers (int, float, complex), strings, tuples are immutable. In addition, custom classes are typically mutable. To ensure immutability and to work with custom classes as data types, we have a number of have a few options. For example, it is possible to use either *namedtuple* from the collections package or *NamedTuple* from the typings package. Another option is to use dataclasses with the *frozen=True* option [13, Chapter7] or to use *BaseModel* from the third party library Pydantic [14]. This has one big advantage over the other methods, Pydantic allows type validation.

```
//example of Typings NamedTuple
class GameBoard(NamedTuple):
   game_board: Tuple[Tuple[Piece, ...], ...]
   currPlayer: str = "R"

//Example of Pydantic BaseModel
class GameBoard(BaseModel):
   game_board: Tuple[Tuple[Piece, ...], ...]
   currPlayer: str = "R"
   class Config:
      frozen = True
```

Figure 7: Example implementation of immutable Classes

Figure 7 shows two examples of *GameBoard* implementations for the game of checkers. One uses a *NamedTuple*, while the other uses the class *BaseModel* of the third-party library Pydantic.

It's worth mentioning another third-party library that provides a robust implementation: pyrsistent. Its functionality is such that when changes are made to our data structure, it generates a new immutable object with the necessary changes, without altering the original structure. It is particularly useful for collection-like data structures such as lists, dictionaries and sets. [15]

RECURSION Python allows Recursion. However due to certain limitations it may not be the most efficient option in most cases. Python is comparatively slow at recursion and has a default stack depth limit of 1000, which can impact the performance of deep recursive functions. Additionally, Python lacks tail call optimization, a feature found in some languages that enhances the efficiency of deep recursion, as discussed in chapter 2.5. However, if the nature of the problem allows for it, such as when employing a divide-and-conquer strategy or dealing with recursive data structures, recursion can be a suitable choice. Despite its drawbacks, recursion can be powerful in specific contexts.[6, Chapter 1]

COMPREHENSIONS AND LAZY EVALUATION Using comprehensions makes code more compact and shifts our focus from the how to the what. The concept comes from Haskell [4]. Simply put, it is an expression that uses the same keywords as loop and conditional blocks, but reverses their order. This inversion helps to emphasise the data rather than the procedure.

```
//Solving without comprehension
collection = list()
for datum in data_set:
   if condition(datum):
      collection.append(datum)
   else:
      new = modify(datum)
      collection.append(new)

//with comprehension
collection_list = [d if condition(d) else modify(d)
for d in data_set]
//with generator comprehension
collection_iter = (d if condition(d) else modify(d)
for d in data_set)
//consume iterator by client function
client = list(collection_iter)
client == collection_iter
// --> True
```

Figure 8: Convert imperative code to comprehension

Figure 8 shows how a comprehension can be used. It is more compact than the imperative code and shifts our thinking to to "what collection is this" and avoids the need to think about "what is the state of the collection at this point in the loop? Comprehensions are widely used in Python. They exist for various data types such as lists, dictionaries, tuples and so on. [6, Chapter1]

LAZY EVALUATION In Python there exists also a Generator comprehension. They have a very similar syntax as shown in figure 8. The big difference with a generator comprehension is that it is lazy evaluated. Such a generator returns an iterator. So they are just a description of "how to get the data", which is not realised until you explicitly ask for it. [6, Chapter1] To better understand this mechanism, a clearer description is needed: Python collections are described as iterables. This means that a for statement can be used to iterate over the values because collections can return an iterator object. An iterator therefore yields a sequence of values that can be consumed by a client function, as shown in 8. So-called generator functions can also be used to create an iterator, but they are not necessarily pure like generator comprehensions. They work like normal functions in Python, but instead of returning a value, we use the value, we use the keyword "yield". [6, chapter3]

5 COMPARISON

The following compares the results of the chapters 3 and 4.

Overall, Python and Go are considered general-purpose programming languages. Python was designed with simplicity in mind, while Go was designed specifically for systems programming. A notable difference between the two lies in their typing systems - Go is statically typed, meaning that variable types are determined at compile time, whereas Python is dynamically typed, meaning that types are determined at runtime. Both languages take a multi-paradigm approach, supporting procedural, object-oriented and functional programming styles. Both Python and Go are classified as impure languages because they allow side effects and mutable state in their programming models.

FIRST CLASS CITIZENS Both languages treat functions as first-class citizens. In Python, everything is an object, including functions, so they are also called first-class objects. Overall, there is no significant difference between Go and Python.

HIGHER-ORDER FUNCTIONS Both languages support higher-level functions. Go does not have the amount of functionality that Python offers. Python has more built-in higher-order functions. In addition, Python has built in the decorator pattern, which is widely used. So, in comparison, Python has more features in the direction of higher-order functions.

IMMUTABILITY In Go, it is important to understand the difference between pass by reference and pass by value in function calls to ensure immutability. By using pass by value, Go automatically works on a copy of the underlying data. One exception is the map data structure. It always behaves like pass by reference. In Python, we can define immutable classes. Python always uses pass by reference, so we have to be careful with mutable data structures such as lists, dictionaries and sets. Third-party libraries such as pyrsistent [15] could help to avoid side effects.

RECURSION Neither language is optimised for recursion. Recursion is slower, has limited stack space, and lacks tail call optimisation in both Go and Python. It should rather be avoided to use recursion. One exception occurs, if it fits the problem very well e.g. if algorithms using recursive data structures like trees or graphs.

LAZY EVALUATION Python has strong support for lazy evaluation due to its generator concept. In particular, generator comprehensions

are very useful because it is a purely functional approach. In general, comprehensions, lazy evaluated or not, are very popular in the Python community. Most Python programmers use them instead of the predefined higher-order functions like map or filter. In Go, it is also possible to write lazy evaluated functions, but the language does not automatically convert the code to them. The programmer itself has to take care of.

PURITY Since Python and Go are impure functional languages, it is possible for them to produce side effects in their function calls, and the functions may also not be idempotent. To achieve as much purity as possible there are some good practices that can be followed, such as avoiding global state or separating pure and impure functionality. Mixing functional programming with the object-oriented paradigm, and hence stateful objects, can also be challenging. After all, both languages contain mutable objects which, if used carelessly, can lead to impure code.

SUMMARY Both languages implement the basics of functional programming in some way, allowing programmers to use functional design patterns and techniques. In summary, Python supports more functional concepts than Go. It seems that functional programming is not as dominant in Go as it is in Python. However, both are multi-paradigm languages that are not designed to be purely functional by nature. Compared to a purely functional language like Haskell, there are a number of drawbacks. For example, both have mutable data structures, allow side effects, have restrictions on recursion, and lack lazy evaluation.

6 CONCLUSION

This thesis shows what the basics of functional programming are and how these concepts are implemented in Python and Golang. Therefore, the basics of functional programming are introduced before the concrete implementation of those in Golang and Python is shown. Comparing the two languages, it appears that Python is more focused on implementing functional features than Go. While Go allows for higher-order functions, function currying, and other basic stuff, Python allows for even more sophisticated advanced techniques such as comprehensions, lazy evaluation, or built-in higher-order functions in its standard library. There are quite a few features that Python has that Go does not.
Go is a relatively new language compared to Python and is still under development. The introduction of generics makes functional pro-

gramming easier and more convenient. Support for other advanced functional features may soon follow, as the example of the slices package [11] shows.

REFERENCES

[1] Susan Haejin Lee. Functional programming is finally going mainstream. https://github.com/readme/featured/functional-programming, accessed on: 02.01.2024.

[2] Checkers. https://en.wikipedia.org/wiki/Checkers, accessed on: 10.01.2024.

[3] Minimax-Algorithmus. https://de.wikipedia.org/wiki/Minimax-Algorithmus, accessed on: 10.01.2024.

[4] Kuchling A. M. Functional Programming HOWTO — Python 3.10.13 documentation. https://docs.python.org/3.10/howto/functional.html, accessed on: 02.01.2024.

[5] *FUNCTIONAL PROGRAMMING IN GO Apply Functional Techniques in Golang to Improve the Testability, Readability, and Security of Your Code*. PACKT PUBLISHING LIMITED, S.l., 1st edition edition, 2023.

[6] David Mertz. *Functional programming in Python*. O'Reilly Media Inc., Sebastopol, Calif., may 2015, first edition edition, 2015.

[7] The Go Programming Language Specification - The Go Programming Language. https://go.dev/ref/spec, accessed on: 10.01.2024.

[8] The Evolution of Go. https://go.dev/talks/2015/gophercon-goevolution.slide#26.

[9] Elliot Chance. Pie. Enjoy a slice! A utility library for dealing with slices and maps that focuses on type safety and performance. https://github.com/elliotchance/pie, 2023.

[10] Berthe Samuel. Lo. A Lodash-style Go library based on Go 1.18+ Generics (map, filter, contains, find...). https://github.com/samber/lo, 2023.

[11] Slices package - slices - Go Packages. https://pkg.go.dev/slices, accessed on 03.01.2024.

[12] Habib Izadkhah and Rashid Behzadidoost. *Challenging Programming in Python: A Problem Solving Perspective*. Springer, Cham, 2024.

[13] Steven F. Lott. *Functional Python Programming: Use a Functional Approach to Write Succinct, Expressive, and Efficient Python Code*. Packt Publishing, Birmingham, third edition edition, 2022.

[14] Welcome to Pydantic - Pydantic. https://docs.pydantic.dev/latest/, accessed on: 02.01.2024.

[15] Pyrsistent — Pyrsistent 0.X.Y (moving target) documentation. https://pyrsistent.readthedocs.io/en/latest/intro.html, accessed on: 03.01.2024.