The Go Programming Language

Moser David
Department of Informatics
University of Zurich
Zurich, Switzerland
david.moser2@uzh.ch

Abstract—Today's programming languages like C, Java or Python are not best fitted for multicore CPU's. The language Go, developed by Google tackles that programming challenge efficiently. Which paradigms it incorporates and how it compares to other programming languages is discussed in this paper. Furthermore, we discuss the implications of Go for the programming landscape of tomorrow.

Keywords—Go, Programming languages, Python, Java, C, interface, concurrency

INTRODUCTION

Designed in 2007, announced in 2009 and released in 2012, Go became a multi-paradigm programming language quickly outgrowing what the initial designers Robert Griesemer, Ken Thompson and Rob Pike at Google would have ever expected [1]. The language was developed by Google as an answer to the impending growth of the number of multi-processor computers. In this paper, we are going to investigate some of the paradigms of Go and compare it to other popular languages like C, Python, or Java. Go is a multi-paradigm, concurrent, object-oriented, statically typed, compiled language that skyrocketed quickly into the lists of top-rated programming languages.

RELATED WORK

The language Go borrows methods from work older than the language itself. Its built-in concurrency heavily depends on the ideas of Hoare's communicating sequential processes [2]. Not being a research programming language, most of related work focuses on real-world application as T. Guney does in [3] or teaching the language like Donovan in [4]. However, the language itself and its history is very well documented on the official website of Go [5].

PARADIGMS

A. Design

The motivation for the development of Go can be best described by the original designers of Go. The Software Engineers at Google at that time mostly used C++ or Java for their servers, both of which take long to compile [6]. Thus, the language was not to be a mere research language but should become useful for production at Google: "the goals of the Go project were to eliminate the slowness and clumsiness of software development at Google, and thereby to make the process more productive and scalable. The language was designed by and for people who write—and read and debug and maintain—large software systems" [6]. The main target of the language can be described to be to "combine the efficacy, speed and safety of a strongly and statically compiled language with the ease of programming of a dynamic language, so as to make programming more fun again" [7].

The first step to reduce compile time was to tackle the inefficient dependency imports of C++, where a header file normally has a "#ifndef" guard so that a file may be included multiple times, by multiple modules. Nevertheless, the compiler opens the files and scans them once for every time it is included in a dependency. This is where Go offers a better solution, where dependencies are defined centrally by the language, meaning the language itself takes care of correct and efficient dependency management.

To achieve efficient dependency management, Go projects contain a central go.mod file which contains the dependencies. This has the effect, that each source is read exactly once. Furthermore, Go does not allow for unused imports; unlike in Java or Python, where unused imports trigger a recommendation to delete the import, Go triggers an actual error, meaning the code will not even compile. [8, 9]

B. Object-Orientation

Even though Go is described as an Object-Oriented language, it does not have classes as it is known in Java or Python. It can be viewed as Object-Oriented, however, as it provides types and methods. However, there exists no type hierarchy or inheritance. The only way to achieve something like type hierarchy is to include one type into a struct-like type, which would be comparable to fields of a class in Java.

What Go provides however, are interfaces. Yet these use a different approach from what is used in Java. In Java, one typically defines an interface as a set of method that a class then can implement, explicitly using the 'implements' keyword [10]. In Go, one specifies an interface type, containing some method specifications [11], typically between zero and three [7]. Instead of explicitly implementing interfaces, all types that implement the methods of a interface implicitly satisfy that interface, making pieces of codes coupled more loosely [6].

That approach enables duck-typing, a feature seen in dynamic languages like Python being a form of polymorphism, yet the compiler still catches obvious mistakes like passing a Boolean where an object method was expected. As an example, consider the following interface declaration:

```
type ReadCloser interface {
   Read(b []byte) (n int, err error)
   Close()
}
```

and a function

```
func ReadAndClose(r ReadCloser, buf []byte)
(n int, err error) {
   for len(buf) > 0 && err == nil {
      var nr int
      nr, err = r.Read(buf)
      n += nr
      buf = buf[nr:]
   }
```

```
    r.Close()
    return n, err
}
```

The first argument of the function ReadAndClose is of the type ReadCloser, which in fact can be any type if this type implements the ReadCloser interface. The type does not have to express the implementation of the interface explicitly, demonstrating the power of Go interfaces. Also, note that any type implementing the ReadCloser interface automatically implements the Reader interface:

```
type Reader interface {
    Read(b[] byte) (n int, err error)
}
```

Furthermore, passing an invalid type, one gets an error at compile time and not, in contrast to Python, at run time. [12]

C. Syntax

With many programmers at Google working with C++, the designers of Go wanted to make the syntax like other C languages, resulting in small education cost for the developers. Some of the main differences regarding the syntax to C are the following:

1) Declarations are backwards

In Go, to order to declare a variable 'a' to be of the type int, we write 'var a int', whereas in C the equivalent would be 'int a;'. For programmers originating from C this might seem strange at first, yet it has clear advantages over the syntax of C. Firstly, it makes it easier to parse and build. Secondly, it reduces confusion about the actual types. Consider the C code 'int* a, b;'. One would expect 'a' and 'b' to be of the same type, however, a is a pointer to an integer, whereas b is an integer. In Go, 'var a, b *int' eliminates that confusion, because now both 'a' and 'b' are pointers. [13]

2) Type inference

In Go, it is possible to use a short declaration syntax with the ':=' operator, e.g., 'a:=5', where 'a' implicitly gets the type int at compile time. Overwriting 'a' with another type is not possible, as Go still is statically typed.

3) Semicolons

Even though semicolons still exist in Go, they are mostly eliminated, as they do not offer a surplus to the programmer but are needed by the compiler. To achieve this, semicolons are still part of the formal grammar, but are injected automatically by the lexer at any line that could be the end of a statement. However, this influences the brace style. This is the reason why codes like

```
if (true) { doThis() }
else { doThat() }
```

becomes invalid (the lexer inserts a semicolon after the first line, terminating the statement block). [14]

4) Ternary Operator

The ternary operator '?:' does not exist in Go for the sake of traditional if-else clauses being more readable and programmers not truly having the need for an identical conditional control flow construct.

5) Unused Variables

The Go language allows neither for unused imports nor for unused variables. Unused variables are a sign for a bug and slow down compilation. The compiler does not produce a warning but an actual error for unused variables.

Other than that, C programmers will have no difficulties getting a hang of Go, as the syntax is close to C.

D. Memory Management

Other than in C, Go provides a built-in automatic garbage collection mechanism. Every programmer who deals with memory management knows that memory management constitutes tremendous overhead, which Go eliminates by solving garbage collection centrally once and for most of the use cases.

E. Concurrency

Traditional languages like C++ and Java were created in a time where the word multi-processor was not ubiquitous as today. Therefore, the languages themselves did not really offer concurrency service at language level. Go introduced a variant of communicating sequential processes (CSP) with first-class channels [2, 7]. It proposes using so-called goroutines and channels. A goroutine is a part of an application that runs concurrently, which is not mapped one-to-one to a thread. It is in fact, much lighter than a thread and use little additional memory. A channel is then used to communicate (send and receive) between goroutines. [7]

F. Error handling

In contrast to Java's try / catch or Python's try / except clauses, Go does not offer this functionality. Instead, typical functions return a second type, an error type, which can then be handled explicitly. This type is an interface and is declared as

```
type error interface {
    Error() string
}.
```

E.g., executing

```
f, err := Sqrt(-1)
```

then returns an error err, which is not nil, but of the type error. We can then check if it is nil and react accordingly. Furthermore, Go does not provide built-in assertions. Both characteristics force the programmer to think about proper error handling and reporting.

DISCUSSION

In the landscape of popular programming languages like C++, Python or Java, Go has risen to be the language of choice for many companies when it comes to programming large systems. Neither the syntax nor the implementation seems to be groundbreakingly new or complex. In fact, the syntax is very close to the syntax of C and the implementation uses ideas from Hoare's CSP or languages designed for concurrency like Newsquek and Limbo [15].

The main reason for its existence is the lacking addressing by C++ and Java for the issues of slow builds, uncontrolled dependencies, programmers only using subsets of the language, duplication of effort, cost of updates, and so on [6]. Rob Pike, one of the founders of Go describes the languages C++ and Java to be 'hard to use', 'subtle, intricate, and verbose', and finds 'patterns' used widely on those languages to be 'a demonstration of weakness in a language'. [16]

Furthermore, these languages are more than 25 years old and designed in a time where Moore's Law was valid for single-core processors. In the last decade, as we have been reaching physical boundaries of atoms, this law could only persist thanks to the rise of multi-core processors [17]. Programming for multi-core processors is challenging and the 'standard' languages did not deliver paradigms for concurrency per se, and are even today poorly adapted to the current clouds of networked multicore CPUs [16].

Go not only offers the possibility to tackle these issues, but also includes the solution in the language itself. Concurrency is solved by built-in goroutines and first-class channels, and the dependency problem is solved by a central dependency management system.

Being a young language, Go first concentrated on the core problems being identified back in 2007, and has implemented other key features like generics later in its maturing process [18]. Having mastered the key functionalities, the language is now ready to implement some new features that are seen in other languages. The growth of Go can be described as exponential, regarding that it skyrocketed to the Top 20 of all language ranking indices [19, 20].

CONCLUSION

The language Go has overrun the world since its initial design in 2007. Its popularity can be summarized by its key features: the static-typing helps prevent runtime errors (unlike in Python) and benefits safety; the simple syntax makes it beginner-friendly and not excessively verbose; it features lightweight concurrency which can be used for building high performance web back-ends; it offers automatic garbage collection, benefiting the programmer; and finally, is pure fun learning.

REFERENCES

[1] Rob Pike, *New Case Studies About Google's Use of Go | Google Open Source Blog.* [Online]. Available: https://opensource.googleblog.com/2020/08/new-case-studies-about-googles-use-of-go.html (accessed: May 2 2022).

[2] C. A. R. Hoare, *Communicating sequential processes*. New York: Prentice Hall, 2000.

[3] T. Guney, *Hands-On Go Programming: Explore Go by solving real-world challenges*, 1st ed. Birmingham: Packt Publishing Limited, 2018. [Online]. Available: https://www.wiso-net.de/document/PKEB__9781789534870176.

[4] A. Donovan and B. Kernighan, *The Go Programming Language*, 1st ed.: Addison-Wesley Professional, 2015.

[5] *The Go Programming Language Specification - The Go Programming Language.* [Online]. Available: https://go.dev/ref/spec (accessed: May 4 2022).

[6] Rob Pike, *Go at Google: Language Design in the Service of Software Engineering - The Go Programming Language.* [Online]. Available: https://go.dev/talks/2012/splash.article (accessed: May 2 2022).

[7] I. Balbaert, *Way To Go: A Thorough Introduction To The Go Programming Language*: iUniverse, 2012.

[8] Rob Pike, *Go at Google: Language Design in the Service of Software Engineering - The Go Programming Language* (accessed: May 4 2022).

[9] *Managing dependencies - The Go Programming Language.* [Online]. Available: https://go.dev/doc/modules/managing-dependencies (accessed: May 4 2022).

[10] J. Bloch, *Effective Java*. Boston, Columbus, Indianapolis, New York, San Francisco, Amsterdam, Cape Town, Dubai, London, Madrid, Milan, Munich, Paris, Montreal, Toronto, Delhi, Mexico City, São Paulo, Sydney, Hong Kong, Seoul, Singapore, Taipei, Tokyo: Addison-Wesley, 2018. [Online]. Available: http://www.amazon.co.uk/dp/0134685997.

[11] J. Meyerson, "The Go Programming Language," *IEEE Softw.*, vol. 31, no. 5, p. 104, 2014, doi: 10.1109/MS.2014.127.

[12] Russ Cox, *Go Data Structures: Interfaces.* [Online]. Available: https://research.swtch.com/interfaces (accessed: May 4 2022).

[13] *Frequently Asked Questions (FAQ) - The Go Programming Language.* [Online]. Available: https://go.dev/doc/faq# declarations_backwards (accessed: May 2 2022).

[14] *Frequently Asked Questions (FAQ) - The Go Programming Language.* [Online]. Available: https://go.dev/doc/faq#semicolons (accessed: May 4 2022).

[15] *Frequently Asked Questions (FAQ) - The Go Programming Language.* [Online]. Available: https://go.dev/doc/faq#What_is_the_purpose_of_the_project (accessed: May 2 2022).

[16] Rob Pike, "Another Go at Language Design," [Online]. Available: https://web.stanford.edu/class/ee380/Abstracts/100428.html

[17] Intel, *Moore's Law and Intel Innovation.* [Online]. Available: https://www.intel.com/content/www/us/en/history/museum-gordon-moore-law.html (accessed: May 4 2022).

[18] *Frequently Asked Questions (FAQ) - The Go Programming Language.* [Online]. Available: https://go.dev/doc/faq#generics (accessed: May 4 2022).

[19] R. K. Eng, "The Eng Language Index - Richard Kenneth Eng - Medium," *Medium*, 08 Jan., 2016. https://richardeng.medium.com/the-eng-language-index-1de9d41aeb67 (accessed: May 4 2022).

[20] IEEE Spectrum, *Top Programming Languages 2021*. [Online]. Available: https://spectrum.ieee.org/top-programming-languages/ (accessed: May 4 2022).