Fundamentals of Programming Languages

Assignment 6

Message-passing Concurrent Programming in Go and Rust

Mestrado em Engenharia Informática
Faculdade de Ciências da Universidade de Lisboa

2024/2025

Ricardo Manuel Sousa Costa Nº 64371

Prof. Vasco Vasconcelos

1 Introduction

Both Rust and Go implementations followed the same design principles using a message-passing approach. The solutions involve three key components:

- **Serialization:** Converting a representation in memory of a JSON object into a stream of tokens (JC) sent over a channel;
- **Deserialization:** Converting a stream of tokens (JC) into a representation in memory of a JSON object;
- **Evaluation:** Applying a accessor to transform a JSON stream into a new stream of tokens.

While the overall architecture is similar in both implementations, there are notable differences in implementation due to the inherent features of both languages.

2 Rust Implementation

Rust leverages major features that were used in the implementation of the solution:

- **Enums**: For the definition of types (`Json`, `JC`, `Accessor`) in a simpler way;
- **Pattern Matching**: For control flow and type safety, that allowed for a more concise and readable code;
- **Thread safety**: Enforced by the compiler through the ownership and borrowing system, which allowed for a more robust and safe implementation.

3 Go Implementation

Go also has some features that helped in the implementation of the solution in different ways:

- **Goroutines**: For concurrent execution of the different components of the solution with lightweight threads;
- **Built-in Channels**: For communication between the different components of the solution;
- **Simplicity**: Go's simplicity and ease of use.

4 Major Decisions Taken

In both implementations, the major decisions taken were related to the design of the types and the control flow of the program, namely in the definition of the streaming tokens and the accessor types. In Rust the types were implemented using enums while in Go types were implemented using structs, which led to some differences in the implementation.

In the `JC` type definitions, there was the need to add control tokens to signal the start and end of an array or object in order to correctly interpret the stream of tokens and convert it into a JSON representation properly. Additionally, there was also the need to a subchannel to send the elements of that array or object in particular, using a `mscp::Receiver` in Rust and a `chan` in Go.

In the `Accessor` type definitions, `Field` was used to represent object property access, `Index` for array index accessing, `Map` for applying accessors to arrays or objects and `End` to signal the end of the accessor chain. Also, in Go, there was added a `Sub` type to chain accessors, which was not needed in Rust since enums can direclty store nested accessor values as part of their definition, which is not possible in Go with structs.

5 Comparison of Implementations

5.1 Rust

5.1.1 Advantages

Rust advantages over Go:

- Strong type and memory safety prevents many bugs at compile time;
- Pattern matching logic makes the code more readable and concise.
- Ownership and borrowing ensure thread safety and avoid data races, with no need for manual synchronization;

5.1.2 Disadvantages

Rust disadvantages over Go:

- Higher complexity due to the ownership and borrowing system, which required a function for cloning and discarding JSON objects;
- Heavier threading model compared to Go's lightweight goroutines.

5.2 Go

5.2.1 Advantages

Go advantages over Rust:

- Lightweight goroutines with a simpler implementation;
- Simpler channels for communication between goroutines.

5.2.2 Disadvantages

Go disadvantages over Rust:

- Lack of enums and pattern matching leads to a more verbose and less readable code;
- More checks at runtime than compile time due to lack of strong type system;
- Worse packaging system and testing tools.

6 Major Difficulties

6.1 Rust

The major difficulties encountered in Rust were related to the ownership and borrowing system, which made it a bit harder to implement the solution, but once the program compiled, it immediately worked correctly, which is the great thing about Rust.

6.2 Go

The major difficulties encountered in Go were related to the lack of enums and pattern matching, which made it a bit harder to define types and the control flow of the program. In Go, some bugs were only found at runtime, which made the debugging process a bit more difficult.

7 Conclusion

Both languages offer effective solutions for this problem with messaging passing concurrency. The choice is a matter of trade-offs of both languages:

- Rust is ideal for applications where type safety, performance and robust error handling are critical;
- Go is ideal for applications where simplicity, ease of use and lightweight concurrency are more important.

Personally, I prefered Rust over Go, mainly because of its strong typing guarantees and syntax, which I find more enjoyable to work with. Also I had less problems and difficulties with Rust than with Go.

8 Running Both Solutions

The steps for running both solutions can be found in the README file.