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

2022/2023

Major decisions

Rust There are some major decisions when approaching the flow of sending both Arrays and Objects. In my JC enum, I added the possibility of sending new extra channels and two new control symbols which control the behaviour of the eval and deserialise.

Whenever we send an Array or an Object in the serialise function, we create a new channel and send the Receiver endpoint to the eval function. This new channel will be exclusive to their content, so when we finish sending it, this auxiliary communication is closed and we proceed to serialise the remainder. To distinguish between Arrays and Objects, we send a control symbol in the same extra channel recently created. If we are dealing with an Array, we send a VS (vector symbol) control symbol, whereas if we are dealing with an Object, an OS (object symbol) is sent instead, allowing us to define two different behaviours in each case.

This describes the communication process between the serialise and the eval functions. Between the eval and the deserialise functions, the logic is the same as described above, adjusted accordingly to the modifications applied by eval on the serialised object, which also decides which elements to forward to the deserialise function.

Furthermore, this demands the existence of two communication channels for the whole program, one for serialise-eval communication and another for eval-deserialise communication. In this solution, I choose to use mpsc channels instead of rendezvous channels.

Go For the solution implemented in Go, I maintained most of the decisions. I adjusted the code to follow Go's characteristics, for example, the inexistence of enums and the difference between their and Rust's channels dynamics.

I also introduced a new channel called quit that informs the main function when each goroutine has finished, making sure they terminate before.

Difficulties

Rust The main difficulties when implementing the Rust solution were related to the ownership of objects in Rust's type system. I came across this problem when trying to serialise each element in an Array or Object. In a loop, at each iteration, I recursively called the serialise function for each element, but since the function's signature requires a Receiver channel endpoint and not its reference, this caused an ownership problem because I was trying to move the same object multiple times.

To bypass this, I found two solutions: to clone the channel or an auxiliary function. I rejected the former to avoid using memory unnecessarily, opting for the latter, which maintains the same signature but requires the Receiver's reference, allowing for the recursive call.

This difficulty appeared later in the eval and deserialise function, but I bypassed it in a similar manner.

Go The Go implementation was a harder challenge due to the abrupt shift from Rust, which I was already comfortable with, to Go.

First, I had issues finding a possible alternative to Rust's enums. Neither structs nor interfaces could completely substitute enums, but I choose structs with some workarounds to adjust their behaviour to my needs. I ended with quite an overly-complex approach which made it difficult to manage and construct JSON objects. I believe there are other ways of approaching this issue, but most of what I tried added more complexity than working with structs only.

Working with channels was a bit of a challenge since they are blocking channels. In the eval function, I had to receive every element sent in the serialise function, even if I didn't use them. Otherwise, the channel would block and I would get undesired results. Rust drops the elements in a channel instead of blocking, so I did not have to deal with this problem in my Rust implementation.

Comparing the solutions

I got the same result in both implementations, but I preferred the solution implemented in Rust.

Rust felt like a much more robust programming language due to the presence of its complex and strong type system, appearing as a much more reliable tool.

Go shines in its simplicity, keeping communications and channel dynamics simple, but does not feel as intuitive as Rust after understanding both. The implementation in Go is more complex due to the lack of a structure equivalent to Rust's enums. Instead, we have structs which demand workarounds in this exercise. The non-blocking Rust channels also remove code complexity, while in Go the channels block if the buffer is full.

How to run each solution

In both implementations I used the JSON object given in the assignment as my example to run and test the implementation. As in for the accessors, I also used those given in

the assignment but left them in comments, so to test the various scenarios it is need to uncomment and comment each accessor accordingly.

Rust To run the Rust implementation we simply have to type the following two commands in the terminal, assuming that Rust is installed in the machine:

```
rustc fc52761_rust.rs
.\fc52761_rust
```

Go To run the Rust implementation we simply have to type the following command in the terminal, assuming that Go is installed in the machine:

```
go run fc52761_go.go
```