Exploring Go's Data Race Detection

Yifei Sun

yifei.sun@utah.edu

May 11, 2023

Abstract

Go has become an increasingly popular programming language, large companies have started using it for critical components in their production systems. However, as a language designed for concurrent programming and with easy access to spawn light-weight threads, even experienced developers can write code with race conditions hidden deep within, making the debugging process time consuming and challenging. The built-in race detection tool in Go may increase memory usage by 5-10x and slow down execution by 2-20x [Aut22a], and as a dynamic data race detector, it can provide relevant warning messages only at runtime. In this project [1], we are exploring the possibilities of running static analysis on Go source code to achieve race detection and provide useful diagnostic information to developers similar to Go's built-in data race detector with relatively low false positive and false negative rate.

1 Introduction

In the realm of concurrent programming languages, Go stands apart with its unique approach to managing communication between concurrently executing processes. The philosophy of Go is encapsulated in the slogan: "Do not communicate by sharing memory; instead, share

[1]GitHub: `https://github.com/StepBroBD/SRD`.

memory by communicating." [Aut22b], this central tenet underpins the design of Go's concurrency features, with channels at its heart. However, like any powerful tool, Go's channels feature is not immune to misuse. A common area of misunderstanding is the proper use of channels in conjunction with goroutines, for example, knowing when to close channels and ensuring that goroutines do not leak by failing to receive from them. Misuse of these features can lead to runtime panics or goroutine leaks, both of which can be problematic in a production environment. Through this project, we hope to demystify these complexities and formalize the correct usage of Go's channels and goroutines using structural operational semantics (SOS) and integrating SOS rules into static analysis tools.

2 Related Work

Type-Based Race Detection for Java [FF00]: The main idea of this paper is to present a static race detection analysis for multithreaded Java programs based on a formal type system that captures many common synchronization patterns. The paper describes the implementation of this system, called rccjava, and demonstrates its effectiveness in eliminating race conditions by checking over 40,000 lines of Java code. The authors also discuss the benefits of using this type system, including early detection of race conditions and improved documentation of locking strategies used by the program.

While the paper specifically focuses on Java programs, the ideas presented in the paper can be applied to other statically typed languages, including Go. It's possible to use similar approach mentioned in the paper to build a race detector that tracks shared variables and checks that they are accessed safely by multiple go routines. It also requires additional annotations to be added to the code to help identify shared variables and synchronization points.

The type system presented in this paper works by tracking the protecting lock for each shared field in a Java program and verifying that the appropriate lock is held whenever a shared field is accessed. This is done through a set of rules that are added to Java's type system, which allow the type checker to reason about the locking discipline used by the program. The type system also requires additional type annotations to be added to the code, which help identify shared variables and synchronization points. These annotations are used by the type checker to ensure that shared variables are only accessed while the appropriate lock is held.

An Operational Semantic Basis for OpenMP Race Analysis [AG17]: This paper presents an operational semantics for OpenMP that models the concurrency structure of OpenMP and enables data race detection for structured parallelism. The authors argue that their approach can help researchers and tool developers better understand OpenMP concurrency, and build more precise data race checkers that reduce memory overheads.

One key idea from the paper that could be applied to Go's data race checking is the use of an operational semantics to model concurrency structures. Other ideas from the paper that could be applicable to Go's data race checking include using state machines to implement operational semantics rules, defining conventions for operational semantics rules, and modeling language constructs in a concurrency model.

Operational Semantics for Multi-Language Programs [MF07]: This paper takes a step towards higher-level models of interoperating systems by abstracting away low-level details of interoperability and focusing on semantic properties like type-safety and observable equivalence. The authors also discuss related work in this area, including formalisms for the semantics of COM and a type-safe intermediate language designed specifically for multi-language interoperation.

Other Resources: In addition to these academic resources, several practical resources provide valuable insights into structural operational semantics and the patterns/practices of Go. These include: 1. "Operational Semantics" (`https://cs.lmu.edu/~ray/notes/opsem/`), 2. "Go Language Patterns" (`http://golangpatterns.info`), 3. "Go Concurrency Patterns" (`https://go.dev/talks/2012/concurrency.slide`), 4. "Advanced Go Concurrency Patterns" `https://go.dev/talks/2013/advconc.slide`), and 5. "Pipelines and Cancellation" (`https://go.dev/blog/pipelines`). These resources offer a wealth of practical examples and best practices that complement the theoretical foundations presented in the academic literature.

3 Problem

A common misuse of Go's channels and goroutines that can lead to data races involves the incorrect assumption that sending to or receiving from a channel implies synchronization between goroutines.

```go
package main

import "fmt"

func main() {
	var data int

	go func() {
		data++
	}()

	go func() {
		fmt.Println(data)
	}()
}
```

Figure 1: Example of a data race caused by unsynchronized access to shared data across multiple goroutines.

```go
package main

import "fmt"

func main() {
	var data int
	done := make(chan bool)

	go func() {
		data++
		done <- true
	}()

	go func() {
		<-done
		fmt.Println(data)
	}()
}
```

Figure 2: Corrected version of Figure 1 using channels to synchronize goroutines and prevent data races.

In Figure 1, two goroutines are spawned. One increments the global data variable, and the other attempts to print the data variable. However, there is no guarantee of the order in which these goroutines will be scheduled to run. As a result, the printed value could be 0 (if the printing happens before the increment) or 1 (if the printing happens after the increment).

This is a classic example of a data race. The idiomatic Go solution to this problem is to use channels to synchronize the goroutines. The sending goroutine can send a signal over the channel when it has finished its operation, and the receiving goroutine can wait for this signal before it begins its operation. In the corrected code (Figure 2), a `done` channel is used to synchronize the two goroutines. The incrementing goroutine sends `true` over the `done` when it has finished its operation. The printing goroutine waits to receive from the `done` before it begins its operation, ensuring that it only prints `data` after the incrementing goroutine has finished.

This use of channels to synchronize goroutines is in line with the Go philosophy of sharing memory by communicating, rather than communicating by sharing memory. Instead of both goroutines accessing the shared data variable with no synchronization (which leads to a data race), the goroutines use a channel to communicate and synchronize their operations, ensuring that the data variable is only accessed when it is safe to do so.

4 Structural Operational Semantics

To formalize the problem and above mentioned fix, we can represent the program as a set of rules that describe how the state of the system changes based on certain conditions. In this case, we want to model the behavior of goroutines and channels in Go, particularly focusing on the synchronization of goroutines via channels.

The configuration of our program can be defined as a tuple (G, M, C) where:

- G is the set of goroutines,
- M is the memory, represented as a mapping from variables to values,
- C is the set of channels, represented as a mapping from channel identifiers to lists of values.

We can use the following SOS rules:

$$\frac{}{(G, M, C) \xrightarrow{\texttt{go f()}} (G \cup \{\texttt{f()}\}, M, C)}$$ **Goroutine Creation**

$$\frac{\texttt{f()} \text{ is } \texttt{data++}}{(G, M, C) \xrightarrow{\texttt{go f()}} (G - \{\texttt{f()}\}, M[\texttt{data} \to M(\texttt{data}) + 1], C)}$$ **Memory Modification**

$$\frac{\texttt{f()} \text{ is } \texttt{done <- true}}{(G, M, C) \xrightarrow{\texttt{go f()}} (G - \{\texttt{f()}\}, M, C[\texttt{done} \to C(\texttt{done}) \cup \{\texttt{true}\}])}$$ **Channel Send**

$$\frac{\texttt{f()} \text{ is } \texttt{<-done}}{(G, M, C) \xrightarrow{\texttt{go f()}} (G - \{\texttt{f()}\}, M, C[\texttt{done} \to C(\texttt{done}) - \{\texttt{true}\}])}$$ **Channel Receive**

$$\frac{\texttt{f()} \text{ is } \texttt{fmt.Println(data)}}{(G, M, C) \xrightarrow{\texttt{go f()}} (G \cup \{\texttt{f()}\}, M, C)}$$ **Print**

Please note that this is a simplified model. In a more comprehensive model, you would want to handle cases like sending to a full channel, receiving from an empty channel, and more complicated goroutine interactions. In addition, you might want to include a scheduler in the state to control the order in which goroutines are run, and to handle the non-deterministic scheduling of goroutines by the Go runtime.

5 Static Analysis

Accompanying this report, there is a demo Go program that usessimple static analysis technique to detect potential data races. The analysis is performed based on the traversal of the Abstract Syntax Tree (AST) of given Go code specified by a command line argument.
The main structures are:

- The State struct, which models the state of a Go program in terms of active goroutines, memory locations that have been written to, and channels that have been used for sending and receiving data.

- The Visitor struct, which is used for traversing the AST. It updates the State according to the different types of nodes encountered in the AST.

The analysis checks for potential data races by looking for memory locations that have been written to more than once, which could indicate a data race if these write operations are not properly synchronized.

However, it's important to note that this analysis is a simple and somewhat naive approach to data race detection. It has several limitations and assumptions that may lead to inaccuracies:

1. All function calls are assumed to be print statements, which simplifies the analysis but is not accurate for general Go code.

2. The analysis assumes that every increment/decrement operation indicates a write to a shared memory location. However, in real Go programs, many of these operations might be operating on local variables that are not shared between goroutines.

3. The analysis doesn't account for the synchronization provided by Go's concurrency primitives like locks or the proper use of channels.

To make this program a reliable tool for data race detection, it would need to be significantly expanded and refined to accurately model the behavior of Go programs, including the semantics of Go's concurrency primitives and the scope and lifetime of variables. The goroutine creation and memory modification rules would need to be more precisely defined and applied only in appropriate contexts. In its current form, it serves as a basic demonstration of how AST traversal can be used for static analysis.

References

[AG17] Simone Atzeni and Ganesh Gopalakrishnan. An operational semantic basis for openmp race analysis. *CoRR*, abs/1709.04551, 2017.

[Aut22a] The Go Authors. Data Race Detector - The Go Programming Language. `https://go.dev/doc/articles/race_detector#Runtime_Overheads`, 2022. [Accessed 25-Feb-2023].

[Aut22b] The Go Authors. Effective Go - The Go Programming Language. `https://go.dev/doc/effective_go`, 2022. [Accessed 04-May-2023].

[CR22] Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 474–489, New York, NY, USA, 2022. Association for Computing Machinery.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. *SIGPLAN Not.*, 35(5):219–232, may 2000.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 3–10, New York, NY, USA, 2007. Association for Computing Machinery.