Ready, Set, Go!
Data Race Detection and the Go Language

Daniel Schnetzer Fava and Martin Steffen

Department of Informatics, University of Oslo, Norway
{danielsf,msteffen}@ifi.uio.no

Abstract

When compared to other synchronization methods (such as locks), channels communication has arguably received lesser attention from the data race checking community. In this talk, we will present a novel data race checker for a concurrent language featuring channel communication as its sole synchronization primitive.

Background

Early on, the concept of race, meaning the competition for access to a shared resource, was explored in the context of channels [12]. Memory models were absent in that setting and, instead, agents were said to race when sending or receiving messages from the same channel "at the same time." Race checking was then conflated often with confluence or determinacy checking [1], with different schedules leading to potential variations in the flow of information through a system.

Today, more often than not, the word *race* is used to denote a *data race*, where threads compete when performing read and write accesses to the same location in shared memory. Because locks have become a very common mechanism for ensuring race-freedom, the contemporary discussion on data race checking is often devoid of the notion of channels; neither does confluence tend to enter the picture. A notable exception is Terauchi and Aiken [17], who give a modern treatment of confluence checking in the context of channel communication. Their work incorporates a form of shared memory, which the authors call *cell channels*. Their treatment, however, does not address data race checking as a first class property but only as a consequence of determinacy. Although related, data-race freedom is different from determinacy and confluence. There exist non-confluent scenarios that are properly synchronized. It is interesting to note, however, that the task of data race checking in the context of channels has received little attention.

Even though mutual-exclusion has dominated the synchronization landscape, languages based on message passing do have a strong footing. Take the Go programming language as an example [7, 2]. Go has gained traction in networking applications, web servers, distributed software and the like. "It prominently features goroutines, which are asynchronous functions resembling lightweight threads, and buffered channel communication in the tradition of CSP [9] (resp. the π-calculus [13]) or Occam [10]. While encouraging message passing as the prime mechanism for communication and synchronization, threads can still exchange data via shared variables. Go's specification includes a memory model [5] which spells out, in precise but informal English, the few rules governing memory interaction at the language level" [3].

Our approach

In prior work, we have given an operational formalization of a relaxed memory model inspired by the Go memory model's specification. We also proved what is called the DRF-SC guarantee

[3], meaning that the memory model behaves *sequentially consistently* when executing data-race free programs. We now propose a data race checker based on our relaxed memory model and the DRF-SC result in particular.

$$\dfrac{(m, !z) \in E_{hb} \qquad E^{r}_{hb} \subseteq E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', !z)\} \cup (E_{hb} - E_{hb} \downarrow_z)}{p\langle E_{hb}, z := v'; t\rangle \parallel m(\!|E^{r}_{hb},\ z{:=}v|\!) \rightarrow p\langle E'_{hb}, t\rangle \parallel m'(\!|\emptyset,\ z{:=}v'|\!)}\ \textsc{R-Write}$$

$$\dfrac{(m, !z) \in E_{hb} \qquad \mathit{fresh}(m') \qquad \begin{array}{l} E'^{r}_{hb} = \{(m', ?z)\} \cup (E^{r}_{hb} - E_{hb} \downarrow_z) \\ E'_{hb} = \{(m', ?z)\} \cup (E_{hb} - E_{hb} \downarrow_z) \cup \{(m, !z)\} \end{array}}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m(\!|E^{r}_{hb},\ z{:=}v|\!) \rightarrow p\langle E'_{hb}, \texttt{let}\ r =\ v\ \texttt{in}\ t\rangle \parallel m(\!|E'^{r}_{hb},\ z{:=}v|\!)}\ \textsc{R-Read}$$

Figure 1: Strong operational semantics augmented for data-race detection

The detector works by recording read- $(m, ?z)$ and write-events $(m', !z)$, where m, m', etc. are unique identifiers. These events are recorded along a variable's memory cells and in thread-local storage. The memory cell associated with a shared variable z takes the form $m(\!|E^{r}_{hb},\ z{:=}v|\!)$ where m is the identifier of the most recent write to z, E^{r}_{hb} is a set holding read events for loads that have accumulated since the most recent write to z, and v is the variable's current value. Threads are equipped with a set E_{hb} which holds information about read- and write-events that are "known" to the thread as having *happened-before*. When a thread attempts to access a memory location, the detector checks whether there exists a concurrent memory event that conflicts with the attempted access—rules R-Write and R-Read of Figure 1. Not shown here are the rules regarding channel communication. Communication carries over happens-before information between threads, thereby affecting synchronization.

In this talk, we will show that the information needed for race checking is contained in the scaffolding of the DRF-SC guarantee proof. Given our experience, we conjecture that, in general, race checkers may be (semi-)automatically derivable from memory models and their corresponding DRF-SC proofs. We should point out, however, that the operational semantics we propose for data race detection is *not* a weak semantics. Apart from the additional bookkeeping, the semantics is "strong" in that it formalizes a memory guaranteeing sequential consistency. Note that, to focus on a form of strong memory is not a limitation. Given that even racy program behaves sequentially consistently up to the point in which the first data-race is encountered, a complete race detector can safely operate under the assumption of sequential consistency.

When it comes to channel communication, our treatment focuses on bounded channels, including synchronous ones. While channels are often used to enforce order between events from different threads (e.g. a send happens before the corresponding receive completes), we revisit the link between channels and locks and discuss how the effects of a channel's boundedness can be used to ensure mutual-exclusion. The pervasiveness of the notion of mutual-exclusion might explain why it is often thought that a data race involves two (or more threads) accessing memory *at the same time*—or, if not at the same time, that there exists an alternate yet equivalent run in which the conflicting accesses can be placed side-by-side. We argue that this interpretation is misleading. Specifically, we show that there exist race conditions in which conflicting accesses are necessarily ordered and necessarily separated in time by other operations. This observation has interesting and subtle consequences relating back to *anomalies* discussed by Lamport in his seminal paper on vector-clocks and distributed systems [11]. In the talk, we will discuss anomalies arising from the fact that the first-in-first-out nature of channels is not fully accounted for in the happens-before relation as defined by the Go memory model's description. We show alternatives for handling the anomalies, but settle with a race detector formalization that is faithful to the Go language specification.

Finally, we compare our approach to modern data race checkers based on vector clocks. In particular, we contrast our work against that of Pozniansky and Schuster, which introduced an algorithm sometimes referred to as DJIT$^+$ [14], and to Flanagan and Freund and their work on FASTTRACK [4]. Such algorithms have influenced the Thread Sanitizer library [15, 16, 8] upon which the current Go race detector [6] is implemented.

References

[1] Robert Cypher and Eric Leu. Efficient race detection for message-passing programs with nonblocking sends and receives. In *Proc. of 7th IEEE Symp. on Parallel and Distributed Processing, SPDP '95*, pages 534–541. IEEE, 1995.

[2] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language.* Addison-Wesley, 2015.

[3] Daniel S. Fava, Martin Steffen, and Volker Stolz. Operational semantics of a weak memory model with channel synchronization. *J. Log. Algebr. Methods Program.*, 103:1–30, 2019.

[4] Cormac Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In Michael Hind and Amer Diwan, editors, *Proc. of 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '09*, pages 121–133. ACM, 2009.

[5] Go language memory model. `https://golang.org/ref/mem`, 2014. Version of May 31, 2014, covering Go version 1.9.1.

[6] Go language race detector. `https://blog.golang.org/race-detector`, 2013.

[7] Go language specification. `https://golang.org/ref/spec`, 2016.

[8] Google Thread Sanitizer library. `https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm`, 2015.

[9] Charles A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[10] Geraint Jones and Michael Goldsmith. *Programming in Occam2.* Prentice-Hall, 1988.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[12] Friedemann Mattern. Virtual time and global states in distributed systems. In Michel Cosnard et al., editors, *Proc. of Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1988.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.

[14] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multi-threaded C++ programs. In *Proc. of 9th ACM Symp. on Principles and Practice of Parallel Programming, PPoPP '03*, pages 179–190. ACM, 2003.

[15] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proc. of Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[16] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler: compile-time instrumentation for Thread-Sanitizer. In Safraz Khurshid and Koushik Sen, editors, *Revised Selected Papers from 2nd Int. Conf. on Runtime Verification, RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 110–114. Springer, 2012.

[17] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 30(5), article 27, 2008.