Algebra-Oriented Proofs for Optimisation of Lockset Data Race Detectors*

Justus Sagemüller, Volker Stolz, and Olivier Verdier

Western Norway University of Applied Sciences, Bergen, Norway
{jsag,vsto,over}@hvl.no

Abstract

Dynamic detectors for potential data races, based on lockset-intersection tracking, are widely used to safety-check concurrent programs. But to make this real-world-feasible, optimisations can be necessary whose validity has so far lacked a solid proof foundation.

We demonstrate that lockset algorithms possess a simple algebraic structure (a commutative monoid), which can simplify the formalisation of optimisations in a proof assistant.

Concurrent programs require synchronisation mechanisms if shared mutable data is to be used safely. Specifically, unrestricted concurrent mutation easily leads to data races [4], causing data corruption which can be disastrous yet hard to detect.

Besides approaches such as Software Transactional Memory, the standard mechanism to avoid this is using *locks*, which can prevent one thread from proceeding until it is safe to do so. This can cause its own problems though, in the most extreme *deadlock* [1] but also simply missed parallelism opportunities. It is thus desirable to minimise the use of locks, but the nondeterministic nature of race conditions requires reliably detecting them, to ultimately avoid them from being possible in production code.

Such detectors come in both static (compile-time) and dynamic (runtime) flavours. We focus on *lockset algorithms* as defined in [5] (in the following called *Eraser*), specifically the *simple lockset algorithm*.[1] The way this is usually presented is as a state machine on top of the running program: for each thread t, the set of currently-held locks is updated as locks are taken or unlocked on that thread. Furthermore, for each shared variable v, a lockset $C(v)$ is kept as the *protecting set*. This starts out as the set of all possible locks, and whenever thread t accesses the variable, it is intersected with the set of lock held by that thread, i.e.

$$C(v) \leftarrow C(v) \cap \text{locks_held}(t).$$

If this intersection ever becomes empty, it means a thread has accessed the variable while not holding any lock that other threads have held to ensure sovereignty over the variable when accessing it, so the empty set indicates a potential race condition.

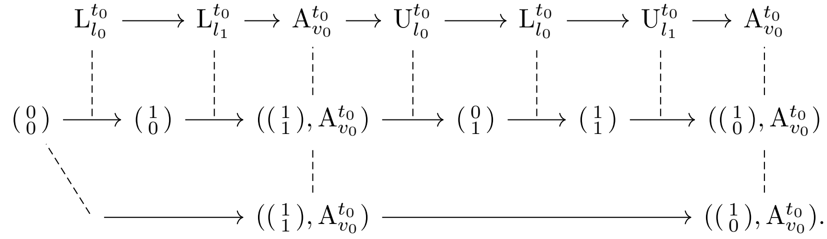
Since the detector does not affect the program flow itself, it can (instead of as a simultaneously-running state machine) as well be formulated to process only a *trace* of the program run[3], where "trace" can be understood as just a stream/list of actions

$$\text{Op} = \begin{cases} \text{Lock } (t : \text{ThreadId})\ (l : \text{LockId}) \\ \text{Unlock } (t : \text{ThreadId})\ (l : \text{LockId}) \\ \text{Access } (t : \text{ThreadId})\ (v : \text{VarId}) \end{cases}$$

*This work was partially supported by the European Horizon 2020 project COEMS under grant agreement no. 732016 (https://www.coems.eu/).

[1] The actual Eraser algorithm has additional states, but these are mostly relevant for initialisation.

Algebraic formulation. For the actual results of the algorithm (i.e. the protecting sets and specifically if they become empty), it is not relevant to consider individual locking or unlocking operations. Rather, it is sufficient to keep track of only the actual lockset states, and even that only at the times when a variable is accessed (because that will be the set which is intersected with the protecting set). So instead of a trace of events, one can consider a trace of accesses together with the lock states: for example, a trace for one thread, one variable, and two locks, is compressed thus:



In this diagram, arrows in the top line represent sequencing of the original events in the trace. In the bottom line, the arrows represent only the transitions between states that actually need to be considered for the race-condition checking.

The vector notation used here already suggests that it is not really necessary to consider the states as sets. The intersection operation that the lockset algorithm applies at each access is but a special case of a *pointwise multiplication*. So generically, these sets can be seen as an instance of an algebra or more precisely a *ring*. Considering only the intersection operation, it is a *monoid*, with the universal set (or, vector-of-all-1-s) as unit. This operation will in the following be written simply as juxtaposition, i.e. $s_a s_b \equiv s_a \cap s_b$.

What the Eraser algorithm does is then left-scanning over the accesses and always applying the held lockset to the right of the protecting set, with the monoid operation:

$$\to \left(\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right), s_0\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\right) \to \left(\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right), s_0\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)\right)$$

so starting from $s_0 = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$, the final state would be

$$\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right) = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right) = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right).$$

For arbitrary locking and access operations, the sequence of relevant checking states becomes

$$(m_0, s_0 m_0) \to (m_1, s_0 m_0 m_1) \to (m_2, s_0 m_0 m_1 m_2) \to \ldots$$

In the general setting of multiple threads, locks and variables, there needs to be one lockset for each thread and one protecting set for each variable, so e.g.

$$\to \big((m_{0,0}, m_{0,1}, \ldots), (\mathrm{A}^{t_1}_{v_0})\big) \to \big((m_{1,0}, m_{1,1}, \ldots), (\mathrm{A}^{t_0}_{v_1})\big) \to \big((m_{2,0}, m_{2,1}, \ldots), (\mathrm{A}^{t_1}_{v_1})\big)$$

is checked as

$$\left((m_{0,0} \ldots), \begin{pmatrix} s_{0,0} m_{0,1} \\ s_{0,1} \\ \vdots \end{pmatrix} \right) \to \left((m_{1,0} \ldots), \begin{pmatrix} s_0 m_{0,1} \\ s_{0,1} m_{1,0} \\ \vdots \end{pmatrix} \right) \to \left((m_{2,0} \ldots), \begin{pmatrix} s_0 m_{0,1} \\ s_{0,1} m_{1,0} m_{2,1} \\ \vdots \end{pmatrix} \right).$$

Effectively, after an access to a variable v, its protecting state is a fold over all previous accesses to v of the lockset state at that time of the corresponding thread.

Computer-assisted proofs. The folding operation lends itself well to a purely-functional implementation, which can be in a language such as Coq:

```
Axiom drcAlgebraic : forall tr v t, let trx := tr ++ [Access t v]
  in drc trx
     <-> match compressTr trx with
         | inl cTr => fold_left pr (map (fun q => fst q (fst (snd q)))
                                        (filter (VId.eqb v ∘ snd ∘ snd) cTr)
                                   ) I <> O
         | inr _ => False
         end.
```

Here, `drc` is an abstract notion of what a data-race checker on a trace of events is. The axiom states that it should be equivalent to our algebraic notion, where `I` is the all-ones state and `O` an all-zeroes state (representing the empty protecting set, i.e. a potential data race). The algebraic form works on the `compressTr` form of the trace. This is actually not defined for arbitrary traces of operations, but only for *well-formed* traces.[2]

We propose using this to prove the correctness of commonly used optimizations that compress or elide information from the trace only based on the underlying monoid. An advantage of the algebraic viewpoint is that it decouples the correctness proofs from any concrete data structure implementing Eraser, as long as an implementation (e.g. through vectors) has the required algebraic properties. For example, we show that repetition of well-formed blocks with balanced locks and unlocks does not add new knowledge about data races, and can hence be elided from the trace:[3]

```
Lemma ast_balanced: forall (pre tr post: list op),
     balanced tr -> wf (pre++tr++post)
  -> forall n, n > 0 -> drc (pre++tr++post) <-> drc (pre++(repeat tr n)++post).
```

This and other equivalences can then be used to avoid the placement of redundant instrumentation (for related work see e.g. [2]) into a system-under-test, and hence through reduced event generation positively affect the computation time for dynamic data race checking.

References

[1] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[2] C. Flanagan and S. N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In G. Castagna, editor, *Proc. of 27th European Conf. on Object-Oriented Programming, ECOOP 2013*, volume 7920 of *Lecture Notes in Computer Science*, pages 255–280. Springer, 2013.

[3] S. Jakšić, D. Li, K. I. Pun, and V. Stolz. Stream-based dynamic data race detection. In *Norsk Informatikkonferanse, NIK 2018*, 12 pp. 2018. Available at http://ojs.bibsys.no/index.php/NIK/article/view/511.

[4] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[5] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[2] A well-formed trace is one that could actually happen in a lock-obeying run of a program. It can not have the same lock taken simultaneously by different threads (nor a lock taken more than once by a single thread).

[3] Without the `post`, `drc` ↔ `drc` is not exactly the same as equivalent lock-info.