Saturn: a library of verified parallelisme-safe data structures for OCaml 5

Clément Allain (INRIA)
Vesa Karvonen (Tarides)
Carine Morel (Tarides)

May 28, 2024

1 Abstract

We present Saturn, a new OCaml 5 library available on `opam`. Saturn offers a collection of parallelism-safe efficient data structures: stack, queue, skiplist, hash table, work-stealing deque, etc. It is well tested, benchmarked and in part formally verified.

2 Motivation

Sharing data between multiple threads or cores is a well-known problem. A naive approach is to take a sequential data structure and protect it with a lock. However, this is often not the best solution. First, if performance is a concern, this approach is likely to be inefficient as locks introduce significant contention. Second, it may not be a sound solution as it can lead to liveness issues such as deadlock, starvation and priority inversion.

In contrast, *lockfree* implementations relying on fine-grained synchronisation rather than locks are typically faster and guaranty system-wide progress. Yet, they are also more complex and come with their own set of bugs: ABA problem (largely mitigated in garbage-collected languages), data races, unexpected behaviors due to non-linearizability.

In this context, Saturn provides a collection of standard lockfree data structures saving OCaml 5 programmers the trouble of designing their own.

3 Library design

Saturn has been designed to meet the needs of programmers looking for parallelism-safe implementations of standard data structures. The choices of data structures proposed in have been made to cover a wide range of use cases, from simple stacks and queues to more complex data structures like skiplists and hash tables. More precisely, Saturn currently features : (A) numerous queues : a lockfree queue based on the well-known Michael and Scott queue algorithm [2], a single-producer single-consumer lockfree queue, a multiple-producers single-consumer queue, and a bounded queue; (B) two stacks : a lockfree stack based on the Treiber stack algorithm [1], and a bounded stack. (C) a lockfree work-stealing deque, (D) a lockfree, bounded bag, (E) a lockfree hash table, (F) a lockfree skiplist.

The implemented algorithms are either well-known, such as the Treiber stack [1], or inspired by such algorithms. Each algorithm was adapted to work with and take advantage of the OCaml memory model. For instance, the Michael-Scott lock-free queue [2] employs a standard algorithm but avoids its memory leak (see the Verification section). (TODO add a note about using DRF-SC memory model for perf).

Regarding performance, Saturn provides benchmarks for each data structure, and significant effort has been dedicated to optimizations. However, there is often a tension

between improving efficiency and remaining within the safe fragment of OCaml 5. In particular, OCaml 5 lacks some important features to avoid false sharing. False sharing occurs when several shared data items are contained in a single cache line. When different domains simultaneously access these data items, they are forced to synchronize, even if they are not modifying the same data. To prevent this, we need to ensure that shared data items are not in the same cache line, which essentially requires padding them. Currently, only atomic variables can be padded (using Atomic.make_contended, available since version 5.2.0), but it is often useful to do more.

The single-consumer single-producer queue provides a good example of this issue:

Add code snippet to show the issue.

There are other missing features in OCaml 5, such as the inability to remove fences on atomic accesses, even when they are redundant with the preceding and/or following atomic accesses. Additionally, there is the issue of the extra indirection in arrays of atomics: each cell in such an array is a pointer to an atomic, which is just another pointer with fences. It is tempting to bypass these current limitations by using unsafe features of the language (e.g., `Obj.magic`). Although these optimizations are performed with care and should not break anything for regular uses, memory safety is not guaranteed for illegal uses and the interaction with compiler optimizations is not clear. As a consequence, we propose two versions of each Saturn's data structures: a safe version and an unsafe version. While most users should find the safe version efficient enough for their needs, daring users may prefer the unsafe version provided they encapsulate it correctly and verify their code somehow.

4 Tests

In multicore programming, it is essential to test not only the correctness of the data structures but also to verify sequential consistency and lock-freedom when expected. Saturn has been thoroughly tested using mainly two different tools: dscheck and STM.

STM is used not only for unit testing but also for verifying sequential consistency. It automatically generates random full programs using the provided API—in the case of Saturn, a data structure. These programs are executed in two domains, and all results are checked based on the post-conditions of each function, providing unit testing. Simultaneously, STM verifies sequential consistency by ensuring that all intermediate states can be explained by a sequential execution of the calls.

dscheck is a model checker : it is designed to compute all possible interleavings of a given program and verify that each one returns the expected result. This is particularly useful for catching elusive bugs that occur only in specific, rare interleavings. Additionally, dscheck can be used to verify that a program is lock-free, as it will fail to terminate if any form of blocking is present.

5 Formal verification

Lockfree algorithms are notoriously difficult to get right. To provide stronger guarantees, we have verified part of Saturn's data structures and hope to verify the entire library in the future. Moreover, one important benefit is that we get formal specifications for verified data structures.

This verification effort has been conducted using Iris [3], a state-of-the-art mechanized *concurrent separation logic*. In particular, all proofs are formalized in Coq.

A common criterion to specify concurrent operations on shared data structures is *linearizability*. The equivalent Iris notion is *logical atomicity*: there exists a point in time when a concurrent operation atomically takes effect (the linearization point). This statement takes the form of an *atomic specification*:

$$\frac{\frac{\frac{\frac{\{\,\text{queue-inv } t\,\}}{\langle\,\forall vs.\,\text{queue-model } t\ vs\,\rangle}}{\texttt{queue\_push}\ t\ v}}{\langle\,\text{queue-model } t\ (vs +\!\!+ [v])\,\rangle}}{\{\,()\,.\,\text{True}\,\}}$$

In this example, we specify the `queue_push` operation from an implementation of a concurrent queue. Similarly to Hoare triples, the two assertions inside curly brackets express the precondition and postcondition. Here, the queue-inv t precondition represents the queue invariant. As it is persistent, we do not need to give it back in the postcondition. The other two assertions inside angle brackets express the *atomic precondition* and *atomic postcondition*. Basically, they specify the linearization point of the operation: during the execution of `queue_push`, the logical model of the queue is atomically updated from vs to $vs +\!\!+ [v]$, in other words v is atomically pushed at the back of the queue.

As a final note, we emphasize that our verification assumes a sequentially consistent memory model. Nevertheless, OCaml 5's relaxed memory model has been formalized [4] in Iris. It should be possible and is future work to adapt our specifications and proofs to support it.

6 Talk proposal

In our talk, we will introduce Saturn, including the main data structures and design guidelines. We will also discuss ongoing work on verifying the library using the Iris concurrent separation logic.

7 Conclusion

By providing extensively-tested, formally verified and benchmarked parallelism-safe implementation, Saturn aims to help OCaml 5 users avoid the pitfalls and intricacies of implementing their own concurrent data structures.

TODO : Add a sentence about what is to come in the future.

References

[1] Robert K. Treiber. *Systems programming: Coping with parallelism.* Tech. rep. RJ 5118. IBM Almaden Research Center, 1986.

[2] Maged M. Michael and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), pp. 267–275. URL: `https://www.cs.rochester.edu/u/scott/papers/1996_PODC_queues.pdf`.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. URL: `https://doi.org/10.1017/S0956796818000151`.

[4] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Cosmo: a concurrent separation logic for multicore OCaml". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 96:1–96:29. URL: `https://doi.org/10.1145/3408978`.