Types for CAS: Relaxed Linearity with Ownership Transfer*

Elias Castegren and Tobias Wrigstad

Uppsala University
`first.last@it.uu.se`

Abstract

This extended abstract overviews work on a type system for lock-free programming based on compare-and-swap. The type system prevents atomicity violations in lock-free programs, where insertion and removal of objects from a linked structure would be subject to data-races breaking linearity of ownership. The type system has successfully been applied to a small number of lock-free data structures.

1 Introduction

Modern hardware is increasingly relying on multi-cores to achieve performance [2]. But with the power of parallelism comes the responsibility of synchronisation – two threads must not be allowed uncontrolled access to the same memory location, as this could lead to data-races and the problems that follow (*e.g.,* lost updates).

On one end of the spectrum we have dynamic synchronisation techniques like locks, where threads have mutually exclusive access to a piece of data, and non-blocking techniques like lock-free algorithms, where each thread follows some protocol in order to guarantee that two operations in conflict cannot both succeed [8]. On the other end of the spectrum we have static techniques like type systems specialised for parallel programming (*e.g.,* [1, 4, 6]).

A very powerful property is the concept of a *linear* (or *unique*) reference, which is a reference that is guaranteed to have no aliases. Like with locks, a thread accessing an object through a linear reference can assume exclusive access, but without the potential runtime overhead of blocking. This however comes at the cost of banning sharing altogether, and requires explicitly passing the reference between threads in order to transfer ownership. This restriction is too strong for many concurrent applications. For example, lock-free algorithms typically require concurrent updates to shared mutable state.

This paper sketches a recently developed type system that uses a relaxed notion of linearity that is powerful enough to guarantee mutual exclusion – a thread always has exclusive access to the linear resources of an object – but at the same time flexible enough to express lock-free data structures where several threads compete to assert ownership of such resources.

Outline § 2 presents the notion of relaxed linearity and ownership transfer and § 3 exemplifies the expressiveness of the system by showing the implementation a Treiber stack [9]. A full treatise of the type system together with more examples, a formalisation and proof of soundness and data-race freedom can be found in our technical report [3].

2 Relaxed Linearity

Traditionally, linear references are references that are statically guaranteed not to have any aliases. They trivially provide mutual exclusion, as holding a linear reference implies holding the *only* reference to an object. Another way to say this is that data-races are only possible if two threads have shared access to some data.

However, aliasing in itself does not necessarily lead to data-races. An important observation is that having several aliases to an object is safe from a data-race perspective as long as at most one of the aliases can be used to access the contents (*i.e.,* the fields) of that object. This is a strong notion of *ownership*, which implies permission to update the object.

*This work is sponsored by the UPMARC centre of excellence, the FP7 project “UPSCALE” and the project “Structured Aliasing” financed by the Swedish Research Council.

Whereas traditional linear types impose a uniqueness restriction on references, we allow unbounded aliasing but require that *ownership* is treated linearly. This allows threads to share objects arbitrarily as long as at most one of the aliases owns the object, and this ownership is never duplicated. Classic linearity equates transferring a reference with transfer of ownership, but we can also transfer ownership between existing aliases. This allows setting up aliasing and later getting the right ownership in place.

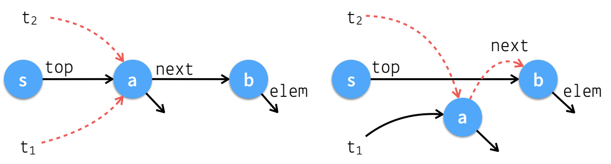

Figure 1: A Treiber stack before and after a successful pop. Black arrows show references with (linear) ownership. Red dashed arrows show non-owning references.

Figure 1 shows a data-structure before and after a transfer of ownership. Black arrows show references with (linear) ownership, and red dashed arrows show references without. Note that no object has more than one black arrow to it. In the left half, `top`, t_1 and t_2 are all aliases, but the latter two have no ownership. In the right half, the ownership in `top` has been transferred to t_1, and the ownership in `a.next` has been transferred to `top`, overwriting its old value. § 3 presents the implementation of the stack that Figure 1 is illustrating.

Further: not all concurrent access patterns to the same memory are harmful. Two threads reading the same memory is trivially safe, but so is having two threads racing to perform an atomic compare-and-swap (`CAS`) to the same memory location a in a (correct) lock-free algorithm. As long the risk of a potential data-race is explicated and its side-effects can be limited, it is safe to have one or more points of contention in a shared data structure.

In our type-driven approach, the programmer must mark the fields of an object which may be subject to (and statically allow) concurrent updates, without requiring that the writer first asserts ownership of the object. Since the value of such a field can change at any point in time, we call the reading of such a field a *speculation*. The field itself is marked by a **spec** keyword. In Figure 1, the `top` field is a **spec** field, meaning it can be updated by any thread

In order to protect the linearity of an object stored in a **spec** field, we require that reads and writes to this field are performed atomically. Specifically, reading a **spec** field creates an alias without ownership. In order to transfer ownership from a **spec** field, the old value must be atomically overwritten using an atomic `CAS` operation. Due to linear ownership, the overwritten field held the *only* ownership of its referenced object, and so a thread can assert ownership of that object after a successful `CAS`. Aliases without ownership can be used for the compare part of the compare-and-swap. In Figure 1, the value in `top` is overwritten, and so its ownership can be transferred into t_1 without duplicating ownership.

The next section introduces our system by example – the implementation of a lock-free stack.

3 A Lock-free Stack

Figure 2 shows the implementation of a Treiber stack [9] in a language using our type system. Each `Node` is linear and contains an element of some elided linear type `T` as well as a reference to the next node in the stack. The `elem` field is protected by linear ownership, meaning there can be at most one reference to a `Node` that has permission to access `elem`. Relaxed linearity however allows the existence of non-owning aliases of a `Node`, which lets threads perform the speculation necessary to perform atomic operations. The `next` field is immutable and can therefore be safely accessed concurrently. Immutable fields are also guaranteed to be stable, so that a value cannot change underfoot. The single point of contention, the `top` field in the stack head, is easily identifiable as it is the only **spec** field in the data structure.

The reads on Lines 13 and 23 give non-owning aliases of the top field of the stack (*cf.,* the red dashed arrows in Figure 1). When the `CAS` in the `pop` function succeeds, ownership is transferred from the `top` field to the variable `t`, which can then be used to destructively read the `elem` field on Line 15. Figure 1 shows how references move and exchange ownership during a successful `pop` operation. Note that t_1 and `s.top` are actually aliases, but the type system makes sure that no thread can extract additional ownership from `t.next` after the `CAS`, even though there might be other threads reading the field (*e.g.,* through `t`$_2$`.next`).

The write to the immutable `next` field on Line 24 is allowed since the object is not visible to other threads yet. As soon as the object is published by the `CAS`, the type system prevents further writes to the field. The resulting ownership transfer of the `CAS` operation in `push` can be understood by reading

Figure 1 from right to left, replacing t_1 by `n` and ignoring the reference t_2 (which cannot exist as the `Node` was created by the current thread and has not yet been shared).

```
struct Stack {
  spec top : Node
}

struct Node {
  var elem : T // var fields are protected
               // by ownership
  val next : Node // val fields are "final"
}

def pop(s : Stack) : T {
  while(true) {
    val t = s.top;
    if (CAS(s.top, t, t.next)) then
      return consume t.elem;
  }
}

def push(s : Stack, e : T) : void {
  val n = new Node;
  n.elem = consume e;
  while(true) {
    val t = s.top;
    n.next = t;
    if (CAS(s.top, n.next, n))
      break;
  }
}
```

Figure 2: A Treiber Stack

Note that the type system preserves linearity of ownership and guarantees that our implementation is data-race free – if a thread manages to pop a `Node`, no other thread can succeed in getting access to the same `Node`. There is no difference in run-time overhead when compared to other standard implementations of the Treiber stack.

4 Conclusions

This paper presented a brief overview of a relaxed notion of linearity that separates aliasing from ownership. Our technical report contains the details of the underlying type system, together with a formalisation of a simple imperative language using it, and proof of soundness and data-race freedom [3]. Apart from the Treiber Stack, we have used the language to implement a Michael-Scott Queue and a Tim Harris List. The type system is non-intrusive as most types can be inferred (Figure 2 compiles and runs in our prototype implementation as is, modulo minor syntactic variations).

Other more powerful verification techniques for concurrent data structures exist (*e.g.,* Separation Logic or Rely-Guarantee References [5]), but we are not aware of other *type systems* aimed at implementing lock-free algorithms. There are type systems that rely on linearity for atomic transfer of ownership (*e.g.,* Rust[7]), but we have not seen this done without using locks or destructive reads, which precludes lock-free implementations.

Our type system captures existing patterns of lock-free programming and enforces correct usage though typing discipline. While we cannot guarantee correctness of a lock-free implementation, with minimum overhead in programmer effort we can exclude data-races and any behavior leading to two threads believing that they own the same value.

References

[1] R. Bocchino. An effect system and language for deterministic-by-default parallel programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[3] E. Castegren and T. Wrigstad. Lolcat: Relaxed linear references for lock-free programming. Technical Report 2016-013, 2016. Uppsala University.

[4] E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP*, 2016.

[5] C. S. Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

[6] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 21–40. ACM, 2012.

[7] N. D. Matsakis and F. S. Klock, II. The rust language. In *HILT*, 2014.

[8] M. Moir and N. Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[9] R. K. Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.