Relaxed Linear References for Lock-free Data Structures[*]

Elias Castegren[1] **and Tobias Wrigstad**[2]

1 **Uppsala University, Sweden,** Elias.Castegren@it.uu.se
2 **Uppsala University, Sweden,** Tobias.Wrigstad@it.uu.se

Abstract

Linear references are guaranteed to be free from aliases. This is a strong property that simplifies reasoning about programs and enables powerful optimisations, but it is also a property that is too strong for many applications. Notably, lock-free algorithms, which implement protocols that ensure safe, non-blocking concurrent access to data structures, are generally not typable with linear references because they rely on aliasing to achieve lock-freedom.

This paper presents LOLCAT, a type system with a relaxed notion of linearity that allows an unbounded number of aliases to an object as long as at most one alias at a time owns the right to access the contents of the object. This ownership can be transferred between aliases, but can never be duplicated. LOLCAT types are powerful enough to type several lock-free data structures and give a compile-time guarantee of absence of data-races when accessing owned data. In particular, LOLCAT is able to assign types to the `CAS` (compare and swap) primitive that precisely describe how ownership is transferred across aliases, possibly across different threads. The paper introduces LOLCAT through a sound core procedural calculus, and shows how LOLCAT can be applied to three fundamental lock-free data structures. It also discusses a prototype implementation which integrates LOLCAT with an object-oriented programming language.

1998 ACM Subject Classification D.3.3 Language Constructs and Features – Concurrent programming structures

Keywords and phrases Type systems, Concurrency, Lock-free programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.6

1 Introduction

In the last decade, hardware manufacturers have increasingly come to rely on scaling through the addition of more cores on a chip, instead of improving the performance of a single core [2]. The underlying reasons are cost-efficiency and problems with heat dissipation. As a result of this paradigm shift, programmers must write their applications specifically to leverage parallel resources—applications must embrace parallelism and concurrency [46, 20].

Amdahl's Law dictates that a program's scalability depends on saturating it with as much parallelism as possible. Avoiding serialisation of execution and contention on shared resources favours lock-free implementations of data structures [40], which employ optimistic concurrency control without the overhead of software transactional memory [26]. Lock-free algorithms are complicated and require that all threads that operate on shared data follow a specific protocol that guarantees that at least one thread makes progress at all times [30].

[*] 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.

Lock-free programming is based on a combination of speculation and publication. For example, when inserting into a lock-free linked list, a thread may speculatively read the contents of some field `x.next`, v, store v in the `next` field of a new node, `n`, and *if* `x.next` *remains unchanged*, publish `n` by replacing the contents of `x.next` by `n`. A key component of many lock-free algorithms is the atomicity of the last two actions: checking if `x.next ==` v and if so, performing `x.next = n`. A common way to achieve such atomicity is through the `CAS` primitive, which is available in modern hardware.

In a lock-free algorithm where several threads compete for the same resource, *e.g.,* access to the same node, care must be taken so that at most one thread succeeds in acquiring it. If two threads successfully extract the same object from a data-structure, subsequent accesses to this object will be subject to data-races—ownership of a resource may not be duplicated.

In the literature on type systems, duplication of ownership is typically prevented using linear references. A linear (or unique) reference is the only reference to a particular object. Linearity is a strong property that allows many powerful operations such as type changes and dynamic object reclassification (*e.g.,* [15]), ownership transfer and zero-copy message passing (*e.g.,* [12, 43, 14]), and safe memory reclamation of objects without garbage collection (*e.g.,* [51]). In the context of parallel programming, linear references do not need concurrency control as a thread holding a linear reference trivially has exclusive ownership of the referenced object (no other thread can even know of its existence) (*e.g.,* [22]). Transfer of linear values across threads without data-races is straightforward.

When programming with linear references one must take care to not accidentally lose linearity [5] as linear values must be threaded through a computation. Most systems maintain linearity through *destructive reads* which nullify variables as they are read [32, 39, 7, 12, 13]. Other systems permit aliasing but additionally require holding a capability (or permission) to allow dereferencing a pointer [45, 25, 52, 37]. To avoid the burden of explicitly chaining linear values through a computation, many systems with linear references allow temporary relaxation of linearity through *borrowing*, which creates a temporary alias that is eventually invalidated, at which point linearity is re-established [4, 12, 25].

Even though a functionally correct lock-free algorithm can guarantee that at most one thread manages to acquire a node in a data structure, linear references and lock-free programming are at odds. Lock-free algorithms generally require an unbounded number of threads concurrently reading from and writing to a data structure, which linear references forbid.

Not only is aliasing a prerequisite of sharing across threads, but using destructive reads to maintain linearity breaks down in the absence of means to write to several locations in an atomic step. Consider popping an element off a Stack implemented as a chain of linear links. A sequential implementation using destructive reads (explicated as **consume**) would perform:

```
Link tmp = consume stack.top;  // Transfer top to the call stack
stack.top = consume tmp.next;  // Transfer top's next to the Stack object
```

A lock-free Stack has contention on its `top` field. Thus, if the `top` field is temporarily nullified to preserve linearity, as in the example above, concurrent accesses might witness this intermediate state and be forced to either abort their operations or wait until the value is instantiated again. Similarly, if access to the `top` field is guarded by some capability, threads must either wait for the capability to become available, or copy the capability and risk overwriting each other's results. Other relaxed techniques such as borrowing are generally not applicable in a concurrent setting as concurrent borrowing of the same object could lead to data-races.

In this paper, we propose a principled relaxation of linearity that separates ownership from holding a reference and supports the atomic transfer of ownership between different aliases to a single object without locks or destructive reads. This enables a form of linear ownership [38] where at any point in time, there is at most one reference allowed to access an object's linear resources. We present a type system, LOLCAT—for *Lock-free Linear Compare and Transfer*, that statically enforces such linear ownership, and use a combination of static and dynamic techniques to achieve effective atomicity of ownership transfer strong enough to express well-known implementations of lock-free data structures, such as stacks [48], linked lists [27] and queues [36]. While our system does not guarantee the correctness of a data structure's implementation with respect to its specification (*e.g.,* it does not guarantee linearizability [31]), it guarantees that all allowed accesses to an object's linear resources are data-race free.

The paper makes the following contributions:

(i) It proposes a linear ownership system that allows the atomic transfer of ownership between aliases (Section 2), with the goal of facilitating lock-free programming and giving meaningful types to patterns in lock-free algorithms, including the `CAS` primitive.

(ii) It shows the design of a type system that enforces linear ownership in the context of a simple procedural language, and demonstrates its expressiveness by showing that it can be used to implement several well-known lock-free data structures (Section 2.4).

(iii) It shows a formalisation of the semantics of a simple procedural language using LOLCAT and proves data-race freedom for accessing linear fields in the presence of aliasing, in addition to type soundness through progress and preservation (Section 3-4).

(iv) It reports on a proof-of-concept implementation (Section 5) in a fork of the object-oriented actor language Encore.

2 Lock-Free Programming with Linearity

This paper presents a principled relaxation of linearity that allows programs whose values are *effectively linear*, although they may at times be aliased, and a hybrid typing discipline that enforces this notion of linearity. Our goal is to enable lock-free programming with the kind of ownership guarantees provided by linear references, and to catch linearity violations in implementations of lock-free algorithms, such as two threads believing that they are the exclusive owners of the same resource.

Our system combines a mostly static approach with some dynamic checks from the literature on lock-free programming (*e.g.,* `CAS`). The latter is needed to avoid data-races when multiple threads read and write the same fields concurrently. Rather than employing advanced program analysis or program logic, we implement our static guarantees as a simple type system, LOLCAT. This design choice trades reasoning power for simplicity and modularity; code can be type-checked locally without the need for interprocedural analysis. This should make it possible or even straightforward to integrate our approach in existing languages.

Our system captures a number of concepts in lock-free programming such as speculation, publication, acquisition and stable paths, and imposes a typing discipline to guarantee their correct usage with respect to linearity. Consequently, we provide a strong notion of ownership in which a pointer (on the stack or on the heap) may own some resources (*i.e.,* values in fields of the object pointed to), and where access to owned resources is guaranteed to be exclusive.

Section 2.1 through Section 2.3 give an overview of the main concepts of LOLCAT. Section 2.4 gives concrete implementation examples.

2.1 The Challenges of Linear Lock-Free Programming

Lock-free programming is complicated, partly due to the lack of mutual exclusion (which *e.g.,* locks can provide). A lock-free algorithm must take into account that values may be accessed and updated concurrently by other threads. This is also the root cause of the challenges one must overcome when designing a type system for lock-free programming:

CHALLENGE 1: *Using linearity to exclude read–write races is too strict as it forces operations to be serialised and allows observation of a data structure in an inconsistent state.*

In the stack popping example from Section 1, we noted that reads of the `top` field of the stack must not consume its value, as this prevents concurrent operations from making progress. Similarly, all threads concurrently pushing to the stack must be able to *simultaneously* alias `top` in the `next` field of a newly created node in each thread, and compete to publish their own node at the head of the stack. Both these requirements break linearity.

We address this challenge by relaxing linearity. At the cost of losing the ability to treat an object's *identity* linearly, we allow *unbounded aliasing of linear values*, as long as each field in the value is accessible through at most one alias. Hence, we have linearity of an object's *fields*, but not its *identity*, similar to systems using permissions (*e.g.,* [45]). To be able to express patterns that appear in lock-free algorithms, we further relax linearity for certain types of fields, and allow these to be accessed through any alias: immutable **val** fields (similar to Java's **final** fields), **once** fields which become immutable after the first write, and **spec** fields which explicitly allow concurrent reading and writing. For consistency, "normal" fields are annotated **var**.

The main guarantee given by relaxed linearity is that accesses to **var** fields are free from data-races. This is ensured by the invariant that a reference ι in a variable or field P is always a *dominator* of the transitive closure C of **var** fields reachable from P. If `f` is a **var** field in C, then any path P' ending in `f` contains ι. If P' is a field access `x.f`, the **var** field `f` is dominated by the stack variable `x`, so the thread holding `x` has exclusive access to the field.

The type system tracks this ownership through the static type `T` of P (`x` in the example above). This is important because no two aliases of P may have static types that allow access to the same **var** field: the reference ι in P owns all **var** fields that its type `T` gives access to.

CHALLENGE 2: *Transferring ownership between aliases, without transferring aliases.*

Lock-free programming requires setting up speculative structures involving aliasing and later attempting to acquire the necessary ownership. Since destructive reads impact other threads' ability to make progress, we must be able to transfer ownership between *existing* aliases rather than equating ownership transfer with alias transfer.

To address this challenge, we employ a novel form of view-point adaptation [41] at the type-level which we term *field restrictions.* These come in three forms: *weak*, *strong* and *transfer*, which all capture existing patterns used in lock-free programming. The intuition of the field restrictions can be explained through a rely–guarantee [34, 44] interpretation:

Weakly restricted types `T|f` guarantee that the field `f` will not be accessed through an alias of this type, and may rely on nothing. This denotes a speculative view of a value, without ownership of the field `f`.

Strongly restricted types `T||f` guarantee that the field `f` will not be accessed through an alias of this type, and may rely on the absence of aliases through which `f` can be accessed. This denotes a view of an object whose field `f` will never be accessed again.

Transfer-restricted types `T` $\sim$ `f` guarantee that an alias of this type will not be used to assert ownership of the object pointed to by the field `f`, and may rely on the fact that the field `f` will not be updated concurrently by another thread. This denotes a view of an object where the field `f` is without ownership, either because it contains a speculation or because ownership has been, or is currently being, transferred from it.

In normal linear type systems, ownership transfer involves moving a unique reference from one place to another, *e.g.,* by using a destructive read. LOLCAT additionally supports ownership transfer through the addition of a field restriction for some ι.`f` in one place and the corresponding removal of a field restriction for the same ι.`f` in another. This allows setting up speculative structures, and also allows transferring ownership from a pointer-based structure without destroying the pointers.

We base this kind of ownership transfer on the atomic compare-and-swap (`CAS`) operation. Even though they are relatively simple, field restrictions let us give a static semantics to the `CAS` primitive that precisely captures how ownership is transferred between aliases when linking and unlinking objects into and out of linked structures (*cf.* Section 2.3).

CHALLENGE 3: *Guaranteeing atomicity of statements that read and write multiple locations.*

The atomic operations used in lock-free programming operate on a single location, yet many lock-free algorithms require operations that modify more than one location without interference. Due to the lack of hardware support for such operations, algorithms must employ clever tricks to achieve "effective atomicity". In a similar fashion, the soundness of our approach, notably the transfer of ownership in Challenge 2, relies on the absence of concurrent modifications of certain fields during operations that atomically move ownership of multiple locations—otherwise, the exclusive access implied by ownership could be compromised.

We solve this problem by leveraging *stable paths*, *i.e.,* fields that are guaranteed not to change and which are therefore accessible without fear of concurrent changes. We support several forms of stable paths: immutable **val** fields; **once** fields which are immutable after initialisation; and *fix pointers* which are pointers that, once installed in a field, cannot be overwritten. As a side-effect of installing a fix pointer in `x.f` where `x` has type `T`, the local type of `x` changes to `T` $\sim$ `f`, which signals that the value in `f` will not change. A dynamic check prevents writes through aliases which are not yet aware that the field has been fixed. See Section 2.5 for an example using fix pointers.

When an object is created, the first reference to it is necessarily globally unique (assuming garbage collection, see Section 5.3). This trivially gives a guarantee that the fields of that object will not change under foot until it has been made accessible to other threads. In LOLCAT, the type annotation **pristine** denotes an object that has just been created, and which is accessible through a single alias only.

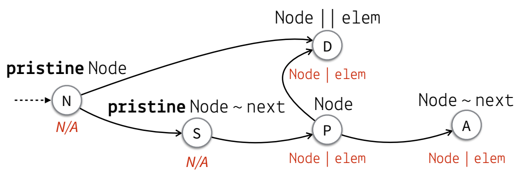
2.2 Typing the Life of a List Node: Speculation, Publication and Acquisition

This section exemplifies some of the concepts of LOLCAT by discussing the implementation of a linked list. In a single-threaded setting this would be simple to implement even using traditional linear types: insertion constitutes creating a new node and linking it in between two existing nodes, and removal is done by unlinking a node. Both operations can be implemented using simple destructive reads as no aliasing is required.

In a multi-threaded lock-free setting, the implementation gets more complicated. Firstly, care must be taken to preserve the integrity of the list in the presence of concurrent accesses

View	Type	Alias' Type
Newborn	**pristine** `Node`	N/A
Staged	**pristine** `Node`∼`next`	N/A
Published	`Node`	`Node` | `elem`
Acquired	`Node`∼`next`	`Node` | `elem`
Speculative	`Node` | `elem`	*any*
Dummy	`Node` || `elem`	`Node` | `elem`

(a) Views of a list node during different stages of its life. The Staged and Acquired views see the `next` field as a reference without ownership. The Speculative and Dummy views may not be used to access the `elem` field.



(b) Transitions between views of a list node during different stages of its life. The types in red (below each view) show which aliases may exist. See Section A for a more detailed version of this figure.

Figure 1 Views, and transitions between views, of a list node with fields **var** `elem` and **spec** `next`.

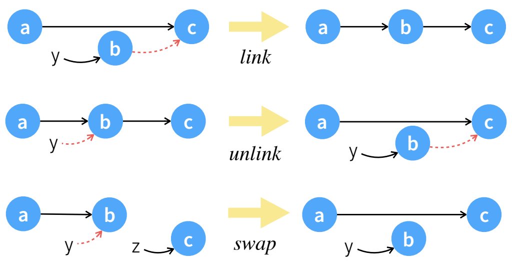
to the `next` fields. Secondly, if two threads were able to unlink the same node, there could be data-races on the value stored in that node. For the reasons brought up in Challenge 1 and Section 1 we cannot solve this problem with traditional linearity, *e.g.,* using destructive reads. In LOLCAT we would implement the list nodes using a type `Node` with a **var** field `elem` storing the element of the link, and a **spec** field `next` of type `Node`. Remember that access to a **var** field is exclusive, while a **spec** field may be accessed concurrently. The **spec** (and **once**) fields of a data-structure identifies the contention points.

A list `Node` goes through several distinct stages during its lifetime. First, the node is created. The LOLCAT type of a newborn node is **pristine** `Node`, where **pristine** captures the global uniqueness of the object. At this point, the thread holding the reference owns the node and may initialise the `elem` field without the risk of data-races.

Second, the intended successor of the new node is speculatively read and written to the `next` field, staging the node for publication (*i.e.,* being inserted into the list). As a side-effect of the field update, the type of the node changes to **pristine** `Node`∼`next` to denote that the `next` field contains a speculation. This means that the current thread still owns the new node, but not the node pointed to by the `next` field. We call writing a speculative value to a field a *tentative write*. During this stage, the `next` field may be written to several times to refresh the speculation, but never dereferenced. The fact that the node is **pristine** (and thus has no aliases) means that no other thread has a view of the node that allows dereferencing or updating `next` concurrently (*cf.,* Challenge 3).

Once the node has been successfully published (*cf.* Section 2.3), its ownership is moved from the stack of the publishing thread to the data structure on the heap. The type of the node internal to the data structure is simply `Node`. The `next` field is no longer restricted as the node now owns its successor, and the node is no longer **pristine** because global uniqueness no longer holds; as mentioned in Challenge 2, any thread accessing the data structure may hold a reference to any of the nodes. The type of such an alias is `Node` | `elem`, capturing that these references are speculations that may not be used to read the `elem` field. Note that once lost, pristineness can never be recovered. This is because we cannot place an upper-bound on the existence of these aliases.

Finally, a thread can manage to acquire ownership of the node and remove it from the list. The type of an acquired node from the view of the acquiring thread is `Node`∼`next`. Since the restriction on `elem` is lifted, this field can be safely read without the risk of data-races.



Top: linking **CAT**, atomically moving ownership of b from y to a and moving ownership of c from a to b.
Middle: unlinking **CAT**, atomically moving ownership of b from a to y and moving ownership of c from b to a.
Bottom: swapping **CAT**, atomically moving ownership of c from z to a and moving ownership of b from a to y.

Figure 2 Different forms of Compare-And-Transfer. Dashed (red) arrows denote references without ownership, *i.e.,* results of speculation or pointers from which ownership has been transferred.

The restriction on `next` captures that `next` points to an object that is owned by someone else (by the data structure on the heap, or by some other thread that have since acquired the successor node). Note that there may still be aliases of the newly acquired node, and that these will still have the type `Node | elem`. This is safe from a data-race perspective as these may not be used to access the `elem` field.

Figure 1 shows how a node's type reflects the view of the stage it is currently in, and which transitions between these views are possible. The type `Node || elem` was not brought up in this example, but reflects permanently burying ownership of the `elem` field (note how there are no edges going out from this state in Figure 1b), turning the node into a "dummy node" that will never access `elem` again. See Section 2.5 for an example where this type is used. There is a more detailed version of Figure 1b in the appendix (*cf.* Section A).

2.3 Atomic Transfer of Ownership

As our main mechanism for transferring ownership between aliases we introduce a **CAT** (compare-and-transfer) operation, which is purposely similar to a `CAS`, but with certain syntactic restrictions. In general, a **CAT** has the form **CAT**`(x.f, p1, p2)`, where `p1` and `p2` are paths of length one or two (*i.e.,* `y` or `y.g`). Like a `CAS`, it *atomically* compares the values of `x.f` and `p1`, and if they are the same, overwrites `x.f` with the value in `p2`. Additionally the types and values of local variables may be updated as detailed below.

The effect of **CAT**`(x.f, p1, p2)` when successful is that ownership is transferred from `p2` to `x.f`, and from `x.f` to `p1`. Remember that LOLCAT guarantees linear ownership of an object's **var** fields; out of all aliases of an object, at most one alias may be used to access the **var** fields of that object. Since `x.f` is overwritten, the transfer of ownership from the original reference to the alias `p1` is safe. Figure 2 overviews the **CAT**s. (The eager reader will find the formal type rules in Figure 12 and implementation details in Section 5.1.)

In the previous section, we saw two examples of ownership transfer: publishing and acquiring a node. The syntactic variant **CAT**`(x.f, n.next, n)` is called a *linking* **CAT** and publishes the necessarily **pristine** node `n` by writing it to `x.f`. It requires that the `next` field is transfer restricted in `n` (**pristine** `Node∼next`) so that it actually contains a speculation that could be an alias of `x.f`. If the **CAT** succeeds, `n` will be implicitly nullified to fully transfer the globally unique node from the publishing thread to the heap. This corresponds to the transition "Staged → Published" of Figure 1b.

Acquiring a node is done with an *unlinking* **CAT** of the form **CAT**`(x.f, n, n.next)`. This transfers `n.next` to `x.f` by overwriting it, and transfers ownership from the newly overwritten reference in `x.f` to `n`. If `x.f` has ownership of a **var** field `elem`, the type of `n` must

```
1  struct Stack {
2    spec top : Node
3  }
4
5  struct Node {
6    var elem : T // T is some elided struct type
7    val next : Node
8  }
9
10 def push(s : Stack, e : T) : void {
11   let n = new Node; // n : pristine Node
12   n.elem = consume e;
13   let t = s.top; // t : Node | elem
14   n.next = t; // n : pristine Node ~ next
15   tryPush(s, consume n);
16 }
```

```
17 def tryPush(s : Stack,
18             n : pristine Node ~ next) : void {
19   if (CAT(s.top, n.next, n)) {
20     // link n between top and next success!
21   } else {
22     let t = s.top; // t : Node | elem
23     n.next = t;
24     tryPush(s, consume n);
25   }
26 }
27
28 def pop(s : Stack) : T {
29   let t = s.top; // t : Node | elem
30   if (CAT(s.top, t, t.next)) { // unlink top
31      // t : Node ~ next
32      return consume t.elem;
33   } else {
34     return pop(s);
35   }
36 }
```

Figure 3 A Treiber Stack with linear nodes and elements. **null**-checks omitted for brevity.

have the `elem` field restricted (`Node | elem`), signaling that it is a speculative value (otherwise the two references could not be aliases). After a successful **CAT**, the restriction on `elem` is lifted from `n` making this reference the new owner of the field. However, since `x.f` now owns the value in `n.next`, `n` must be marked to show that the field is without ownership via the type `Node`∼`next`. This corresponds to the transition "Published → Acquired" of Figure 1b.

If `n.next` could change concurrently while performing **CAT**`(x.f, n, n.next)`, this could lead to inconsistencies in the list as well as duplicated ownership. As mentioned in Challenge 3, there is no hardware support for atomically comparing `x.f` and `n` *and* dereferencing `n.next`. For this reason, the unlinking **CAT** requires that `n.next` is a stable field, either by being a **val** or a **once** field, or by having a fix pointer installed. Section 2.5 has an example of the latter.

Finally, a *swapping* **CAT** of the form **CAT**`(x.f, n1, n2)`, can be used to switch a node on the heap for a node on the stack. Like a linking **CAT**, it consumes (nullifies) the owning reference `n2` on success, and like an unlinking **CAT**, the ownership in `x.f` is transferred to `n1`. Even though the nodes referred to by `n1` and `n2` switch owners, the views of the nodes remain the same. Thus, there is no corresponding transition in Figure 1b.

2.4 LOLCAT in Action: Implementation of a Treiber Stack

Figure 3 shows an implementation of a lock-free Treiber stack [48] in a simple procedural language using LOLCAT. The stack data structure is constructed of two data types, `Stack` and `Node`. Stack "objects" hold a reference to a linked chain of nodes in its `top` field. In a Treiber stack, multiple threads may *read and write* the `top` field concurrently. In LOLCAT, `top` must therefore be marked as speculatable using the **spec** field modifier (Line 2).

Stack nodes in Figure 3 have two fields: **var** `elem:T` and **val** `next:Node`. The `elem` field is a mutable field containing an element pushed onto the stack. The `next` field is immutable, meaning that a node's next node is fixed for life after publication.

Our relaxed linearity allows stack and node objects to be aliased freely, but guarantees that for each node there may be at most one alias that can read its element field—all other aliases must have type `Node | elem`. Because `top`'s type is `Node`, it is guaranteed to hold the only pointer to the top node through which its element is accessible. The same holds for the remainder of the stack because of the type of the `next` field is also `Node`. To enforce that the only way to obtain an element in the stack is to first acquire the node holding it, we only allow variables as targets of field accesses; the `elem` field can only be read after storing a node into a local variable.

Pushing—Speculation & Publication. Pushing an element onto the Treiber stack is implemented by the two functions `push` (Lines 10–16) and `tryPush` (Lines 17–26). In a real programming language with loops, these would have been a single, much shorter, function. We rely on recursion instead of loops in order to simplify the formalism in Section 3

The `push` function creates a new node `n` from the element argument and the current value of `top`. The type of `n` is **pristine** `Node`, which means it is not (yet) visible to other threads. With this knowledge, we can safely allow writes to immutable **val** fields (somewhat similar to constructors writing **final** fields in *e.g.,* Java) repeatedly, until the object is no longer pristine.

Line 13 performs a speculative read of `top`. Speculative reads copy references without transferring ownership. This is visible as variable `t` on Line 13 has type `Node|elem`, which means a node whose element field is inaccessible. Note that the **spec** field `s.top` has type `Node`, meaning it does own the `elem` field. All reads of **spec** fields are speculative—they do not transfer ownership, but create an alias to which ownership can later be transferred.

The assignment `n.next = t` on Line 14 is a tentative write. Although the `next` field in the `Node` struct has type `Node`, we are allowed to store `t` in it, even though `t` is a speculation and does not have the required ownership of `elem` (visible from it's type `Node|elem`). This prima facie type-violating field update is allowed—and sound—for two reasons:

1. It requires that we transfer-restrict `next` in `n`'s type, so that no ownership can be transferred from `next`. This happens as a side-effect of the assignment in LOLCAT.
2. Since `n` is pristine, we know that there are no aliases to `n`, meaning that the type change from `Node` to `Node∼next` is a strong update.

To obtain ownership of the object pointed to by `n.next`, the current thread must succeed in overwriting the source of the speculation, `s.top`, while `s.top == n.next` holds (*i.e.,* perform a successful **CAT**). This will allow the restriction on `n`'s type to be lifted, so that aliasing this object with `Node` as its static type is sound.

The function `tryPush` takes a node `n` of type **pristine** `Node∼next` and attempts to replace the current `top` by `n`. If it fails, it will re-read `top`, update `n.next` with the new value, and re-attempt to replace `top` by `n`. Lines 22–24 are identical to lines 12–14 in `push`.

The pivotal line in `tryPush` is Line 19. It employs a linking **CAT** (*cf.* Section 2.3) to attempt to push the node onto the stack. **CAT**`(s.top, n.next, n)` should be interpreted as "if no other thread has pushed or popped since we speculatively read `s.top` (*i.e.,* `s.top == n.next` holds), transfer ownership from `n` to `s.top` and from `s.top` to `n.next`". If successful, the **CAT** will consume (nullify) `n`, transferring its ownership from the call stack of `tryPush` to the `top` field of the stack data structure on the heap.

Popping—Acquisition. Popping elements off the stack is less involved than pushing them onto the stack. The function `pop` speculatively reads the current value of `top` and then employs an unlinking **CAT**, the dual version of the **CAT** in `tryPush`, to remove the node from the linked structure. **CAT**`(s.top, t, t.next)` should be read as "if no other thread has pushed or popped since we speculatively read `s.top` (*i.e.,* `s.top == t` holds), transfer ownership from `t.next` to `s.top` and from `s.top` to `t`". The unlinking **CAT** requires `n.next`

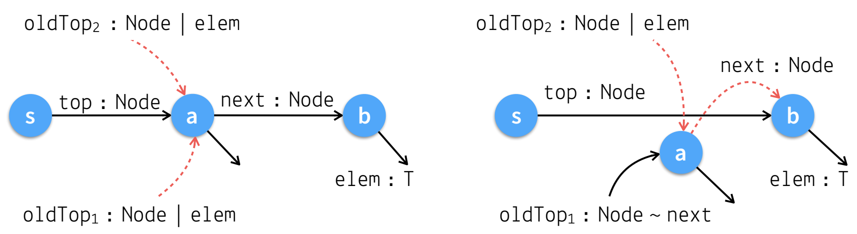


Figure 4 A Treiber stack before and after a successful pop.

to be a stable path (*cf.* Section 2.3), which is true by construction as `next` is a **val** field in `Node`.

Notably, the transfer of ownership from `t.next` to `s.top` preserves the reference in `t.next`. Thus, there are two aliases to the same object, both with type `Node` which seemingly breaks linear ownership. However, on success, the type of `t` is changed to `Node`∼`next` which captures that `t.next` does not own its value, statically preventing using `t` to obtain an owning reference through `next`. Since `t` owns `elem` (otherwise the field would have been restricted in its type), `t.elem` may be destructively read and returned on Line 32, without risking data-races.

Any alias `t'` of `t` in another thread will have the type `Node|elem` and can therefore not access the element field. Since ownership has been transferred from the heap, there is no way for these threads to subsequently acquire ownership of the node just popped: since `s.top` has changed value, **CAT**`(s.top, t', t'.next)` will fail until a thread manages to perform the **CAT** with an up-to-date speculation of `s.top` in `t'`.

Element Ownership. Figure 4 shows a Treiber stack before (left) and after (right) a successful pop, focusing on the ownership of the elements. On the left, s.`top` owns a.`elem`, and a.`next` owns b.`elem`. The types, `Node|elem` of the two `oldTop` references prevent both `oldTop`s from accessing any `elem` fields. On the right, `oldTop`$_1$ holds the unlinked node and thus owns a.`elem`. Although a.`next` is not touched by the operation, it has lost its ownership of b.`elem` to s.`top`. This is tracked at the type level by updating the type of `oldTop`$_1$ to `Node`∼`next`. This is consistent with the global view of next fields as **val**—unlike **spec** fields like `top`, their ownership cannot be directly extracted by overwriting them using a **CAT**.

Summary. The Treiber stack example demonstrated **spec** fields and speculative reads, **val** fields and stable paths, **pristine** values and tentative writes, and how different operations impose or lift weak restrictions and transfer restrictions to preserve linear access to fields. It also exemplified the two dual variants of the compare-and-transfer operation used for publication and acquisition.

An important observation is that all three arguments to a **CAT** have the same type (modulo restrictions) meaning it is tailored for recursive data structures. Although a **CAT** involves multiple operations, the required restrictions on its arguments ensure that it is always possible to implement using a single `CAS` with effective atomicity guaranteed.

2.5 Data Structures with Multiple Contention Points

As demonstrated by the previous example, linking and unlinking nodes in a LIFO stack can rely on the inherent stability of **val** fields to avoid modification of nodes concurrent with unlinking. This is possible because there is only a single point of contention in the data

```
1  def delete(l : List, key : int) : T {
2    let (left, right) = search(l, key);
3    if ((right == l.tail) || (right.key != key))
4      return null; // key does not exist, abort
5    else if (!isStable(right.next))
6      if (fix(right.next)) // Try to fix the field
7        if (CAT(left.next, right, right.next)) // Try to unlink right
8          return consume right.elem;
9        else
10         search(l, right.key); // Someone else came first. Try to help
11   return delete(l, key); // Something went wrong, retry
12 }
```

Figure 5 Harris-style linked list (Excerpt [10])

structure. To support data structures with multiple points of contention, we apply one of the two other techniques for achieving stability mentioned under Challenge 3 in Section 2.1:

Fix Pointers References that cannot be overwritten. Storing a fix pointer into a field effectively makes that field stable. Fix pointers can be implemented with a mark-bit à la `next` pointers in a Tim Harris linked list [27]. The operation **fix**`(x.f)` creates a fix pointer from the reference in `x.f` and subsequently installs it in the same field, returning **true** or **false** depending on if the operation succeeds or not. Section 5.1 discusses implementation.

Once Fields Fields that can only be assigned once, after which they remain constant. They are similar to Java's final fields (and LOLCAT's **val** fields), except that threads may race on their initialisation. We implement **once** fields using fix pointers. We use a **try** operation to write to **once** fields which implicitly creates a fix pointer and which may fail due to concurrent writes from other threads.

While **once** fields can be replaced by a principled use of **spec** fields and fix pointers, they also capture programmer intent in a clear way. A programmer can dynamically check for the presence of a fix pointer using the predicate **isStable**. On a successful branch on **isStable**`(x.f)` or **fix**`(x.f)`, the type `T` of `x` is updated to `T` ∼`f` to reflect our knowledge that `x.f` is stable.

Figure 5 shows an excerpt of a Harris-style linked list [27] (full code is in the technical report [10]) with one point of contention for each node. Inserting a node in a Harris-style list is similar to the Treiber stack, but the possibility of concurrent modification of a node's `next` field during its unlinking (in contrast to the stack, where `next` fields were always **val**) greatly complicates unlinking. To overcome this problem, Harris introduces a logical deletion step, in which a node is rendered immutable by setting a low bit in its `next` pointer, causing subsequent `CAS` operations on this field to fail. We mimic this design using fix pointers in Figure 5. When `right` points to the node to be unlinked, we make sure it is not already logically deleted by checking if it is fixed (Line 5), and then try to **fix** it ourselves (Line 6).

In a Michael–Scott queue [36], there are three points of contention: the `first` and `last` pointers in the queue head, and the `next` pointer of the last node. For this data structure, **once** fields are a perfect match, as they guarantee stability after initialisation, but allow many threads to race to initialise the field in an enqueue operation. We show an implementation of a Michael–Scott queue in Figure 6. Note that an empty queue contains a single dummy node.

Enqueuing to a Michael–Scott queue is similar to pushing to a Treiber stack, with the difference that the new node is appended rather than prepended. The **try** operation on Line 19 of Figure 6 attempts to write the new node to the `next` field of the last node. On success,

```
1  struct Node {
2    var elem : Elem;
3    once next : Node
4  }
5
6  struct Queue {
7    spec first : Node || elem;
8    spec last : Node | elem
9  }
10
11 def enqueue(q : Queue, x : Elem) : void {
12   let n = new Node;
13   n.elem = consume x;
14   tryEnqueue(q, consume n);
15 }
16
17 def tryEnqueue(q : Queue, n : pristine Node) :
18   let oldLast = q.last;
19   if (try(oldLast.next = n)) {
20     // Success, try advance last pointer, return
21     CAT(q.last, oldLast, oldLast.next);
22   } else { // help by advancing last, then retry
23     CAT(q.last, oldLast, oldLast.next);
24     tryEnqueue(q, consume n);
25   }
26 }
27 def newQueue() : Queue {
28   let q = new Queue;
29   let dummy = new Node;
30   q.first = consume dummy;
31   q.last = this.first;
32   return q;
33 }
34
35 def dequeue(q : Queue) : Elem {
36   let oldFirst = q.first;
37   if (isStable(oldFirst.next)) {
38     // oldFirst.next has been written to.
39     // Try to advance first
40     if (CAT(q.first, oldFirst,
41             oldFirst.next) => elem) {
42       return consume elem;
43     } else {
44       // Someone else dequeued before us, retry
45       return dequeue(q);
46     }
47   } else {
48     // oldFirst.next has not been written to.
49     // Retry or fail (here, fail)
50     return null;
51   }
52 }
```

Figure 6 Michael–Scott queue.

a **CAT** is used to advance the `last` pointer. If the write fails, the **once** field has already been written to, and the same **CAT** tries to help global progress by advancing the `last` pointer. In both branches, we know that `oldLast.next` is stable, and so we change the type of `oldLast` from `Node` | `elem` to `Node` | `elem`∼`next`.

Finally, we get to demonstrate the use of strong field restrictions in the type of `first`, *i.e.*, `Node` || `elem`. Dequeuing from a Michael–Scott queue involves swinging the `first` pointer forward to point to `first.next`, making the new first node the new dummy node and extracting the element from it. Because `first.next`'s type is `Node`, `first.next` is the only pointer with ownership of `first.next.elem`. When `first.next` is stored in `first`, this ownership is lost, making the `elem` field globally inaccessible. To avoid this, a **CAT** is able to preserve aliases of otherwise lost fields if they are strongly restricted in the target. We call this *residual aliasing*, and it is shown on Line 41 of Figure 6 as `=> elem`. This introduces a variable `elem` which aliases the field of the same name in the node that was written to `q.first`.

While the types of `first` and `last` differ, the fields alias when the queue is empty. This is fine, as neither type grants ownership of the `elem` field. Also note that variables and/or fields with overlapping strong restrictions cannot alias because each alias could be used to create residual aliases of the same field.

Figure 7 shows an overview of our three example data structures. The labels on the arrows show the fields' modifiers and types. The legend shows what features of our system are exercised by the example. Thick purple arrows show contended fields. Only the **once** field in the node in `last` is contented in the Michael–Scott queue.

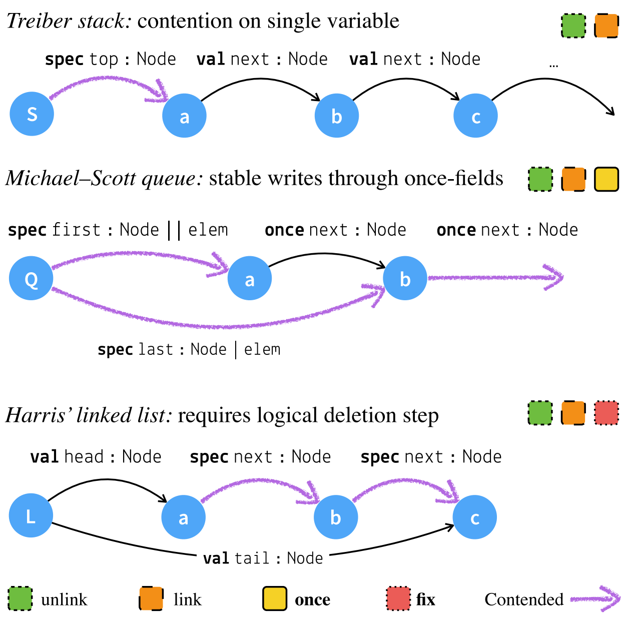


Figure 7 Concepts exercised in the examples.

3 Formalising Linear Ownership in LOLCAT

This section formalises the static and dynamic semantics of a simple procedural language using LOLCAT. Without loss of generality, we exclude "normal references" and consider all references linear. Our implementation of LOLCAT is in an object-oriented language (*cf.* Section 5).

Figure 8 shows the syntax. A program P is a sequence of structs (à la C) and functions followed by an initial expression. Structs are named sequences of fields. A field has a modifier, a name and a type. s and f ranges over names of structs and fields. There are four modifiers on fields that control how a field's content may be modified and shared across threads: **var** fields are mutable and unshared; **val** fields are immutable and shared; **spec** and **once** fields are mutable and shared. A **once** field may be written once. Read–write races are only possible on **once** and **spec** fields. Writes to such fields may fail under contention.

Types are constructed from structs. A type can be **pristine**, denoting a globally unaliased value. Types may have weak and strong field restrictions, and transfer restrictions. The meta variable $\mathtt{T}$ ranges over all types and the meta variable $\mathtt{t}$ ranges over non-pristine types.

Expressions are values (including locations in the dynamic semantics, where they are also subscripted by static types to simplify proofs), paths (variable accesses or field accesses), destructive reads of paths, field updates, creation of new values, function calls, forking of new threads, let-expressions and conditionals. Without loss of generality we restrict functions to a single parameter. More parameters can be encoded using an extra object indirection.

Conditionals branch on boolean expressions which mostly deal with contended writes to fields which may possibly fail due to concurrent modifications: **CAT** publishes and/or acquires values; **try** attempts to install a value in a **once** field; **fix** attempts to write a fix pointer into a **spec** field; **isStable** allows dynamically checking if a field has been fixed.

For simplicity, we formalise our system with let bindings instead of sequences and a flow-sensitive type system, using the standard trick of encoding sequences $e_1; e_2$ as **let** $\_ = e_1$ **in** e_2. Consequently, **CAT**, **fix** and **try** must be used as guards of conditionals, and we reflect changes

$$
\begin{array}{rcll}
P & ::= & \overline{S}\,\overline{F}\,e & \textit{(Program)}\\
S & ::= & \textbf{struct}\ s\ \texttt{\{}\ \overline{Fd}\ \texttt{\}} & \textit{(Struct)}\\
Fd & ::= & \mathit{mod}\ f : \texttt{T} & \textit{(Field)}\\
\mathit{mod} & ::= & \textbf{var} \mid \textbf{val} \mid \textbf{once} \mid \textbf{spec} & \textit{(Modifier)}\\
F & ::= & \textbf{def}\ \mathit{fn}(x : \texttt{T}) : \texttt{T}\ \texttt{\{}\ e\ \texttt{\}} & \textit{(Function)}\\
\texttt{T} & ::= & \textbf{pristine}\ \texttt{t} \mid \texttt{t} & \textit{(Type)}\\
\texttt{t} & ::= & s \mid \texttt{t}\,|\,\texttt{f} \mid \texttt{t}\,||\,\texttt{f} \mid \texttt{t}\sim\texttt{f} & \textit{(Struct type)}\\
e & ::= & v_{\texttt{T}} \mid p \mid \textbf{consume}\ p \mid \textbf{new}\ s \mid x.f = e \mid \mathit{fn}(e) \mid & \\
 & & \textbf{fork}\ \mathit{fn}(e); e \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid \textbf{if}\ b\ \texttt{\{}\ e\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e\ \texttt{\}} & \textit{(Expression)}\\
p & ::= & x \mid x.f & \textit{(Path)}\\
v & ::= & \iota \mid \textbf{null} & \textit{(Value)}\\
b & ::= & \textbf{CAT}(x.f, e, e) \Rightarrow z \mid \textbf{try}(x.f = y) \mid \textbf{fix}(x.f, y) \mid \textbf{isStable}(x.f) & \textit{(Boolean Expr.)}
\end{array}
$$

Figure 8 Syntax of LOLCAT. We write $\overline{x}$ to mean "many x".

$$\boxed{\vdash P \quad \vdash S \quad \vdash Fd \quad \vdash F} \qquad \textit{(Declarations)}$$

$$
\textsc{wf-program}\ \frac{\vdash \overline{S} \quad \vdash \overline{F} \quad \epsilon \vdash e : \texttt{T}}{\vdash \overline{S}\,\overline{F}\,e}
\qquad
\textsc{wf-struct}\ \frac{\vdash \overline{Fd}}{\vdash \textbf{struct}\ s\ \texttt{\{}\ \overline{Fd}\texttt{\}}}
\qquad
\textsc{wf-field}\ \frac{\vdash \texttt{T} \quad \textbf{safeOnHeap}\,(\mathit{mod}, \texttt{T})}{\vdash \mathit{mod}\,f : \texttt{T}}
\qquad
\textsc{wf-function}\ \frac{x : \texttt{T}_1 \vdash e : \texttt{T}_2}{\vdash \textbf{def}\,\mathit{fn}(x : \texttt{T}_1) : \texttt{T}_2\ \texttt{\{}\ e\texttt{\}}}
$$

Figure 9 Well-formed declarations

of ownership in the types differently in the different branches. When unused, we don't write out the residual alias ($\Rightarrow z$) of a **CAT**. We also rely on recursion instead of loops. These decisions were made to simplify the presentation, and are not necessary for the soundness of the approach. For example, by employing a simple data flow analysis, we could omit several of the local destructive reads necessary to reflect type changes.

3.1 Static Semantics

Declarations (Figure 9). The well-formedness definitions are straightforward (WF-PROGRAM, WF-STRUCT, WF-FIELD and WF-FUNCTION). The only unusual premise is found in WF-FIELD—the predicate **safeOnHeap** that prevents fields' types to be pristine or have transfer restrictions. Additionally, **val** and **once** fields may not be strongly restricted. The details can be found in the technical report [10].

Types and Field Lookup (Figure 10). *Top left:* The type s denotes a value which is an instance of struct s Any well-formed struct type can be **pristine**. Types can additionally have weak or strong restrictions on **var** fields, and transfer restrictions on non-**var** fields.

Top right: The relation $\vdash \texttt{T} \rightsquigarrow \texttt{T}'$ denotes that a value of type $\texttt{T}$ can flow (be assigned) into a field or variable of type $\texttt{T}'$. A type $\texttt{t}_1$ can flow into $\texttt{t}_2$ if all fields which are restricted in $\texttt{t}_1$ are also restricted in $\texttt{t}_2$ (FLOW-*-L). Notably, a value with a strongly restricted field can only flow into a variable where the same field is *weakly* restricted (FLOW-STRONG-L). We use $|f \in \texttt{t}$ to mean "f is weakly restricted in $\texttt{t}$" and similarly for the other restrictions. For arbitrary restrictions we write $f \in \texttt{t}$. By FLOW-R/S, a non-restricted type can always flow into an additionally restricted version of itself. (We write $\_f$ to mean $|\texttt{f}$, $||\texttt{f}$, or $\sim\texttt{f}$.) A pristine type can flow into another pristine type (FLOW-PRIST-PRIST), and pristineness can be forgotten if the underlying types are flow-related (FLOW-PRIST).

$\boxed{\vdash \texttt{T}}$ *(Well-formed type)*

$$\text{T-STRUCT}\ \frac{\mathcal{S}\,(s) = \overline{Fd}}{\vdash s} \qquad \text{T-P}\ \frac{\vdash \texttt{t}}{\vdash \textbf{pristine}\,\texttt{t}}$$

$$\text{T-WEAK}\ \frac{\vdash \texttt{t} \quad \mathcal{F}(\texttt{t}, f) = \textbf{var}\, f : \texttt{T}}{\vdash \texttt{t} |\, f} \qquad \text{T-STRONG}\ \frac{\vdash \texttt{t} \quad \mathcal{F}(\texttt{t}, f) = \textbf{var}\, f : \texttt{T}}{\vdash \texttt{t} \parallel f}$$

$$\text{T-TRANSFER}\ \frac{\vdash \texttt{t} \qquad \sim f \notin \texttt{t} \qquad \mathcal{F}(\texttt{t}, f) = \mathit{mod}\, f : \texttt{T} \qquad \mathit{mod} \neq \textbf{var}}{\vdash \texttt{t} \sim f}$$

$\boxed{\vdash \texttt{T} \rightsquigarrow \texttt{T}'}$ *(Type flow)*

$$\text{FLOW-WEAK-L}\ \frac{|f \in \texttt{t}' \qquad \vdash \texttt{t} \rightsquigarrow \texttt{t}'}{\vdash \texttt{t}|\, f \rightsquigarrow \texttt{t}'} \qquad \text{FLOW-STRONG-L}\ \frac{|f \in \texttt{t}' \qquad \vdash \texttt{t} \rightsquigarrow \texttt{t}'}{\vdash \texttt{t} \parallel f \rightsquigarrow \texttt{t}'}$$

$$\text{FLOW-TRANSFER-L}\ \frac{\sim f \in \texttt{t}' \qquad \vdash \texttt{t} \rightsquigarrow \texttt{t}'}{\vdash \texttt{t} \sim f \rightsquigarrow \texttt{t}'} \qquad \text{FLOW-R}\ \frac{\vdash s \rightsquigarrow \texttt{t}}{\vdash s \rightsquigarrow \texttt{t}\_\ f} \qquad \text{FLOW-S}\ \frac{}{\vdash s \rightsquigarrow s}$$

$$\text{FLOW-PRIST-PRIST}\ \frac{\vdash \textbf{pristine}\,\texttt{t} \rightsquigarrow \texttt{t}'}{\vdash \textbf{pristine}\,\texttt{t} \rightsquigarrow \textbf{pristine}\,\texttt{t}'} \qquad \text{FLOW-PRIST}\ \frac{\vdash \texttt{t} \rightsquigarrow \texttt{t}'}{\vdash \textbf{pristine}\,\texttt{t} \rightsquigarrow \texttt{t}'}$$

$\boxed{\mathcal{F}(\texttt{T}, f) = \mathit{mod}\ f : \texttt{T}'}$ *(Field lookup)*

$$\text{LKUP-F-WEAK}\ \frac{f \neq g \qquad \mathcal{F}(\texttt{t}, f) = \mathit{mod}\, f : \texttt{T}}{\mathcal{F}(\texttt{t}|\, g, f) = \mathit{mod}\, f : \texttt{T}} \qquad \text{LKUP-F-STRONG}\ \frac{f \neq g \qquad \mathcal{F}(\texttt{t}, f) = \mathit{mod}\, f : \texttt{T}}{\mathcal{F}(\texttt{t} \parallel g, f) = \mathit{mod}\, f : \texttt{T}}$$

$$\text{LKUP-F-TRANSFER-EQ}\ \frac{\mathcal{F}(\texttt{t}, f) = \mathit{mod}\, f : \texttt{T}}{\mathcal{F}(\texttt{t} \sim f, f) = \textbf{val}\, f : \texttt{T}} \qquad \text{LKUP-F-TRANSFER-NEQ}\ \frac{f \neq g \qquad \mathcal{F}(\texttt{t}, f) = \mathit{mod}\, f : \texttt{T}}{\mathcal{F}(\texttt{t} \sim g, f) = \mathit{mod}\, f : \texttt{T}}$$

Figure 10 Typing and selected field lookup ($\mathcal{F}$) rules.

Bottom: A weakly or strongly restricted field cannot be accessed at all (LKUP-F-WEAK), (LKUP-F-STRONG). A transfer restricted field appears stable (LKUP-F-TRANSFER-*). For brevity, we relegate some cases of field from Figure 10 to the technical report [10].

Expressions (Figure 11). To keep track of the static types of locations in the dynamic semantics, we subscript values with the static type of the expression from which they were reduced. For example, if x has static type $\texttt{T}$, and holds **null** at run-time, we write $\textbf{null}_{\texttt{T}}$ in the program under reduction. Type subscripts are only used to simplify the proofs, and do not affect the semantics of a program.

As usual, **null** can have any valid type (E-NULL). A location is well-typed if its dynamic type can flow into its subscripted (static) type (E-LOC). Typing locations in a program under reduction is only used in the meta-theory. Linear variables can be read non-destructively if the type is not pristine and all **var** fields are forgotten in the resulting type (E-VAR). We use the helper function **restrict**($\texttt{T}$) to restrict all **var** fields in a type $\texttt{T}$, preserving the linear ownership of any **var** fields in x. Similarly, fields can be read non-destructively if all **var** fields are forgotten in the resulting type (E-SELECT). By design, **once** fields cannot be read directly, but must first be checked to have a value using **isStable**($x.f$). This restricts the field, making it appear as an (accessible) **val** field (B-STABLE) (*cf.,* Figure 13).

Destructively reading a variable or field transfers its value to the stack of the current thread. As the values are transferred, they are not restricted (E-CONSUME-VAR, E-CONSUME-FD). By design, destructive reads are only available on **var** fields and always succeed.

Values are created from well-formed struct declarations and start in a pristine state (E-NEW). A value remains pristine until written to the heap (*i.e.,* it is published).

$\boxed{\Gamma \vdash e : t}$ *(Expressions)*

E-NULL
$$\dfrac{\vdash \mathtt{T} \qquad \vdash \Gamma}{\Gamma \vdash \mathbf{null}_{\mathtt{T}} : \mathtt{T}}$$

E-LOC
$$\dfrac{\Gamma(\iota) = s \qquad \vdash s \rightsquigarrow \mathtt{T} \qquad \vdash \Gamma}{\Gamma \vdash \iota_{\mathtt{T}} : \mathtt{T}}$$

E-VAR
$$\dfrac{\Gamma(x) = \mathtt{t} \qquad \vdash \Gamma}{\Gamma \vdash x : \mathbf{restrict}\,(\mathtt{t})}$$

E-SELECT
$$\dfrac{\begin{array}{c}\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \\ \mathcal{F}\,(\mathtt{T}_x, f) = \mathit{mod}\, f : \mathtt{T}_f \\ \mathit{mod} \notin \{\mathbf{var}, \mathbf{once}\}\end{array}}{\Gamma \vdash x.f : \mathbf{restrict}\,(\mathtt{T}_f)}$$

E-CONSUME-VAR
$$\dfrac{\Gamma(x) = \mathtt{T} \qquad \vdash \Gamma}{\Gamma \vdash \mathbf{consume}\, x : \mathtt{T}}$$

E-CONSUME-FD
$$\dfrac{\begin{array}{c}\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \\ \mathcal{F}\,(\mathtt{T}_x, f) = \mathbf{var}\, f : \mathtt{T}_f\end{array}}{\Gamma \vdash \mathbf{consume}\, x.f : \mathtt{T}_f}$$

E-NEW
$$\dfrac{\vdash s \qquad \vdash \Gamma}{\Gamma \vdash \mathbf{new}\, s : \mathbf{pristine}\, s}$$

E-UPDATE
$$\dfrac{\begin{array}{c}\Gamma(x) = \mathtt{T}_x \\ \mathcal{F}\,(\mathtt{T}_x, f) = \mathbf{var}\, f : \mathtt{T}_f \\ \Gamma \vdash e : \mathtt{T} \qquad \vdash \mathtt{T} \rightsquigarrow \mathtt{T}_f\end{array}}{\Gamma \vdash x.f = e : \mathtt{T}_x}$$

E-UPDATE-PRISTINE
$$\dfrac{\begin{array}{c}\Gamma(x) = \mathbf{pristine}\, \mathtt{t}_x \qquad \mathcal{F}\,(\mathtt{t}_x, f) = \mathit{mod}\, f : \mathtt{T}_f \\ \mathit{mod} \in \{\mathbf{val}, \mathbf{spec}\} \qquad \Gamma \vdash e : \mathtt{T} \qquad \vdash \mathtt{T} \rightsquigarrow \mathtt{T}_f\end{array}}{\Gamma \vdash x.f = e : \mathbf{pristine}\, \mathtt{t}_x}$$

E-UPDATE-TENTATIVE
$$\dfrac{\begin{array}{c}\Gamma(x) = \mathbf{pristine}\, \mathtt{t}_x \qquad \mathcal{F}\,(\mathtt{t}_x, f) = \mathit{mod}\, f : \mathtt{T}_f \\ \mathit{mod} \in \{\mathbf{val}, \mathbf{spec}\} \qquad \Gamma \vdash e : \mathtt{T} \qquad \nexists\, g\ .\ \sim g \in \mathtt{T} \\ \nexists\, g\ .\ \parallel g \in \mathtt{T} \qquad \nexists\, g\ .\ \parallel g \in \mathtt{T}_f \qquad \mathtt{T}_f \neq \mathtt{T} \qquad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}\end{array}}{\Gamma \vdash x.f = e : \mathbf{pristine}\, \mathtt{t}_x \sim f}$$

E-IF
$$\dfrac{\begin{array}{c}\Gamma \vdash b \dashv \Gamma' \\ \Gamma' \vdash e_1 : \mathtt{T} \qquad \Gamma \vdash e_2 : \mathtt{T}\end{array}}{\Gamma \vdash \mathbf{if}\,(b)\, \{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\} : \mathtt{T}}$$

E-CALL
$$\dfrac{\begin{array}{c}\mathcal{P}\,(\mathit{fn}) = (x : \mathtt{T}_1, \mathtt{T}_2, e_2) \\ \Gamma \vdash e_1 : \mathtt{T}_1\end{array}}{\Gamma \vdash \mathit{fn}(e_1) : \mathtt{T}_2}$$

E-FORK
$$\dfrac{\begin{array}{c}\mathcal{P}\,(\mathit{fn}) = (x : \mathtt{T}_1, \mathtt{T}_2, e_2) \\ \Gamma \vdash e_1 : \mathtt{T}_1 \qquad \Gamma \vdash e : \mathtt{T}\end{array}}{\Gamma \vdash \mathbf{fork}\, \mathit{fn}(e_1); e : \mathtt{T}}$$

E-LET
$$\dfrac{\begin{array}{c}\Gamma \vdash e_1 : \mathtt{T}_1 \\ \Gamma, x : \mathtt{T}_1 \vdash e_2 : \mathtt{T}_2\end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \mathtt{T}_2}$$

Figure 11 Well-typed expressions. $\mathcal{P}$ denotes function lookup.

As **var** fields are only accessible to one thread at a time, access is data race-free. The resulting value of a field update $x.f = e$ is the target x, which is consumed in the process (E-UPDATE). By binding the result in a **let**-expression we can track type changes to the target (see below). With a fully flow-sensitive type system, such a trick would not be necessary.

Pristine targets allow updating **val** and **spec** fields without the use of a **CAT** (E-UPDATE-PRISTINE). Since pristine values are unaliased, updates to a **val** field are not visible to other threads, and writes to **spec** fields are uncontended. We are allowed to assign a weakly restricted value into an unrestricted field to perform a tentative write (E-UPDATE-TENTATIVE). This causes a strong update of the target that restricts the field written to, which prevents unsoundly extracting an owning alias of the speculative value. We are however allowed to publish the pristine object with a linking **CAT**, *overwriting the source of the speculation*. This confirms the validity of the speculation and lifts the restriction on the field (*cf.*, B-CAT-LINK in Figure 12). To maintain the property that a strongly restricted field is globally inaccessible, we disallow tentative writes when either type involved has any strongly restricted fields[1].

[1] This is strictly not necessary since the field written to will be transfer restricted, which keeps the value inaccessible. However, showing this is complicated, and there doesn't seem to be much to gain from allowing it.

$\boxed{\Gamma \vdash b \dashv \Gamma'}$ *(Compare and transfer)*

B-CAT-LINK

$$\frac{\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \textbf{pristine}\,\mathtt{t}_y \sim g \qquad \mathcal{F}(\mathtt{T}_x, f) = \textbf{spec}\,f : \mathtt{T}_f \qquad \mathcal{F}(\mathtt{t}_y \sim g, g) = \textbf{val}\,g : \mathtt{T}_g \qquad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_g \qquad \vdash \mathtt{t}_y \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \textbf{CAT}\ (x.f, y.g, y) \dashv \Gamma}$$

B-CAT-UNLINK

$$\frac{\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \mathtt{T}_y \qquad \mathcal{F}(\mathtt{T}_x, f) = \textbf{spec}\,f : \mathtt{T}_f \qquad \mathcal{F}(\mathtt{T}_y, g) = \textbf{val}\,g : \mathtt{T}_g \qquad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_y \qquad \vdash \mathtt{T}_g \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \textbf{CAT}\ (x.f, y, y.g) \dashv \Gamma[y \mapsto \mathtt{T}_f \sim g]}$$

B-CAT-SWAP

$$\frac{\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \mathtt{T}_y \qquad \Gamma(z) = \mathtt{T}_z \qquad \mathcal{F}(\mathtt{T}_x, f) = \textbf{spec}\,f : \mathtt{T}_f \qquad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_y \qquad \vdash \mathtt{T}_z \rightsquigarrow \mathtt{T}_f}{\Gamma \vdash \textbf{CAT}\ (x.f, y, z) \dashv \Gamma[y \mapsto \mathtt{T}_f]}$$

B-CAT-RESIDUAL

$$\frac{\Gamma(x) = \mathtt{T}_x \qquad \mathcal{F}(\mathtt{T}_x, f) = \textbf{spec}\,f : \mathtt{T}_f \qquad \|\, g \in \mathtt{T}_f \qquad \Gamma \vdash \textbf{CAT}\ (x.f, p_1, p_2) \dashv \Gamma' \qquad \Gamma \vdash p_2 : \mathtt{T}_2 \qquad \mathcal{F}(\mathtt{T}_2, g) = \textbf{var}\,g : \mathtt{T}_g}{\Gamma \vdash \textbf{CAT}(x.f, p_1, p_2) \Rightarrow z_g \dashv \Gamma'[z_g \mapsto \mathtt{T}_g]}$$

Figure 12 Compare and transfer.

For simplicity, we propagate type changes through **if** statements (E-IF). With a fully flow-sensitive type system operations such as writing to **once** fields could appear anywhere, as the field will be stable regardless of whether the write succeeds or not. The type rules for boolean expressions b are found in Figure 12 and Figure 13. The else branch of **if** statements always maintains the environment.

Function calls, forking and let-bindings are straightforward.

Compare and Transfer (Figure 12). Compare and transfer comes in three forms (*cf.*, Figure 2): link (**CAT**`(x.f,y.g,y)`) inserts an object in a chain of links; its dual, unlink (**CAT**`(x.f,y,y.g)`) removes an object from a chain; swap (**CAT**`(x.f,y,z)`) trades places of whole trees dominated by the arguments of the **CAT**. To highlight these differences, we describe each form in a separate type rule.

On success, **CAT** operations may modify the environment by lifting restrictions on **var** fields in local variables involved in the **CAT**, or by adding residual aliases. Residual aliases are otherwise lost as a side-effect of strong field restrictions on the value being transferred. For simplicity, we consider only a single residual alias, whose type is inferred from the types involved in the **CAT**. For example, if transferring a value of type `T` into a field of type `T || f`, the residual alias be the value of the f field.

By B-CAT-LINK, inserting an object o to create a chain of links $o_1.f \to o.g \to o_2 \cdots$ requires that o is pristine and that its g field is transfer restricted. The requirement that it is pristine guarantees that the g field is not modified concurrently. The restriction requirement ensures that g actually contains a speculation, and prevents using o to obtain an owning reference from $o.g$ (*cf.*, E-UPDATE-TENTATIVE). The field f where o is inserted must be a **spec** field and have a type that o can flow into when the transfer restriction on g is lifted.

By B-CAT-UNLINK, unlinking the object o from the chain above requires that its g field is stable (note that transfer restricted **spec** and **once** fields appear as **val** fields) and that the target is a **spec** field with a type that $o.g$ can flow into. A successful transfer installs an owning reference to o in y, but with the g field transfer restricted. This allows keeping the reference in $o.g$ to avoid confusing other threads accessing o concurrently, but prevents violating linearity by using y to turn $o.g$ into an owning reference.

The rule for swapping two owning references, B-CAT-SWAP, corresponds to a common CAS, except that we require the target field to be explicitly denoted speculatable.

$\boxed{\Gamma \vdash b \dashv \Gamma'}$ *(Fix pointers and once fields)*

$$\textsc{B-TRY}\quad \frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \textbf{pristine}\,\mathtt{t}_y \\ \mathcal{F}(\mathtt{T}_x, f) = \textbf{once}\, f : \mathtt{T}_f \qquad \vdash \mathtt{t}_y \rightsquigarrow \mathtt{T}_f\end{array}}{\Gamma \vdash \textbf{try}\ (x.f = y) \dashv \Gamma[x \mapsto \mathtt{T}_x \sim f]}$$

$$\textsc{B-FIX}\quad \frac{\begin{array}{c}\vdash \Gamma \qquad \Gamma(x) = \mathtt{T}_x \qquad \Gamma(y) = \mathtt{T}_y \\ \mathcal{F}(\mathtt{T}_x, f) = \textbf{spec}\, f : \mathtt{T}_f \qquad \vdash \mathtt{T}_f \rightsquigarrow \mathtt{T}_y\end{array}}{\Gamma \vdash \textbf{fix}\ (x.f, y) \dashv \Gamma[x \mapsto \mathtt{T}_x \sim f]}$$

$$\textsc{B-STABLE}\quad \frac{\vdash \Gamma \qquad \mathcal{F}(\mathtt{T}_x, f) = \mathit{mod}\, f : \mathtt{T}_f \qquad \Gamma(x) = \mathtt{T}_x \qquad \mathit{mod} \in \{\textbf{once}, \textbf{spec}\}}{\Gamma \vdash \textbf{isStable}\ (x.f) \dashv \Gamma[x \mapsto \mathtt{T}_x \sim f]}$$

Figure 13 Operations on fix pointers and **once** fields.

By B-CAT-RESIDUAL, a successful **CAT** will produce a residual alias from a strongly restricted field whose value would otherwise be lost. For example, transferring a pointer ι with ownership over $\iota.g$ holding v into some field whose type strongly restricts g would lead to the program globally losing access to v in the program. Thus, v can be "saved" as a residual alias ($\Rightarrow z_g$ in the figure).

Fix Pointers (Figure 13). Writes to **once** fields must be performed using **try** and placed in an **if** statement to handle both possible outcomes (success and failure). After a successful write to a **once** field, we update the type of the target to prevent further writes to the field by the current thread (B-TRY). This restriction means field lookups will make the field appear as a **val** field, which is needed for the linking and unlinking **CAT**s. If the write fails, the field is also stable as it is already written to (*cf.*, Section 2.5). For simplicity we omit that type change in the formalism, as adding a call to **isStable** in the **else** branch gives the same result. Even though the type change is only visible in the first branch of the **if** statement, having an unrestricted alias is fine as subsequent attempted writes will fail. While writes to **once** fields are discernible through the target's type, we use specialised syntax to highlight that its semantics is different from a normal assignment (which always succeeds).

A speculatable field can be fixed, which causes all future writes to it to fail (B-FIX). Since fix pointer creation involves a contended write, we require a witness of the intended value. Fixing the pointer will succeed if the witness is equal to the field. Like with **try**, a successful **fix** changes the type of x to a type where f is transfer restricted. The same type change occurs when checking if a field has a fix pointer installed (B-STABLE).

3.2 Dynamic Semantics

A configuration is a triple $\langle H; V; T\rangle$. H is a heap mapping locations ι to structs (s, F), where s is the type of the struct and F is a map from field names to values. V is a map from variables to values and their static types. The types of structs and variables are only recorded to simplify meta-theoretic reasoning and do not affect the semantics of a program. T is a list $e_1 || \ldots || e_n$ of expressions running in parallel, that never block and can step at any time.

To simplify the meta-theoretic reasoning, we subscript values on the stack with their static type. Values on the heap are subscripted by $\phi ::= \epsilon \mid *$ which captures whether a reference is a fix pointer ($*$) or may be overwritten (ϵ). This corresponds to a Harris-style mark bit in a pointer [27].

$\boxed{cfg \hookrightarrow cfg'}$ *(Dynamic semantics)*

$$\textsc{d-var}\quad \frac{V(x) = v_{\mathtt{T}}}{\langle H;\, V;\, x\rangle \hookrightarrow \langle H;\, V;\, v_{\mathbf{restrict}\,(\mathtt{T})}\rangle} \qquad \textsc{d-consume-var}\quad \frac{V(x) = v_{\mathtt{T}}}{\langle H;\, V;\, \mathbf{consume}\, x\rangle \hookrightarrow \langle H;\, V[x \mapsto \mathbf{null}_{\mathtt{T}}];\, v_{\mathtt{T}}\rangle}$$

$$\textsc{d-consume-fd}\quad \frac{V(x) = \iota_{\mathtt{T}} \qquad H(\iota) = (s, F) \qquad F(f) = v_{\phi} \qquad \mathcal{F}(\mathtt{T}, f) = \mathit{mod}\, f : \mathtt{T}'}{\langle H;\, V;\, \mathbf{consume}\, x.f\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto \mathbf{null}_{\epsilon}])];\, V;\, v_{\mathtt{T}'}\rangle}$$

$$\textsc{d-select}\quad \frac{\begin{array}{c} V(x) = \iota_{\mathtt{T}} \\ H(\iota)(f) = v_{\phi} \qquad \mathcal{F}(\mathtt{T}, f) = \mathit{mod}\, f : \mathtt{T}' \end{array}}{\langle H;\, V;\, x.f\rangle \hookrightarrow \langle H;\, V;\, v_{\mathbf{restrict}\,(\mathtt{T}')}\rangle} \qquad \textsc{d-new}\quad \frac{\iota\ \mathbf{fresh} \qquad \mathcal{S}(s) = \overline{\mathit{mod}_i\, f_i : \mathtt{T}_i}^{\,n}}{\langle H;\, V;\, \mathbf{new}\, s\rangle \hookrightarrow \langle H, \iota \mapsto (s, \overline{f_i \mapsto \mathbf{null}_{\epsilon}}^{\,n});\, V;\, \iota_{\mathbf{pristine}\, s}\rangle}$$

$$\textsc{d-update}\quad \frac{V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota) = (s, F) \qquad \mathtt{T}' = \mathbf{updateReturnType}\,(\mathtt{T}_x, f, \mathtt{T})}{\langle H;\, V;\, x.f = v_{\mathtt{T}}\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{\epsilon}])];\, V[x \mapsto \mathbf{null}_{\mathtt{T}_x}];\, \iota_{\mathtt{T}'}\rangle}$$

Figure 14 Dynamic Semantics 1/2 (Uncontended operations).

The amount of branching to deal with success and failure of contended operations makes the dynamic semantics surprisingly large for such a small language. In this submission, we therefore relegate the less interesting rules (**let** bindings, function calls, parallelism, etc.) to the technical report [10].

To track local type changes in the branches of **if** expressions, we employ a dynamic variable substitution scheme. The expression ${}^{x:\mathtt{T}}[e]$ should be read as "e with the type of x changed to T". The details can be found in the technical report [10].

Uncontended Operations (Figure 14). The rules D-VAR and D-SELECT show that variables and fields may be read *non-destructively*, creating an alias with a restricted type. Destructively reading a variable or field preserves linearity. The rules D-CONSUME-* show how the source variable or field is nullified as a side-effect of a consume. Note that destructively reading a field is uncontended because the static semantics requires that the target is an owning reference. By D-NEW, new objects are **pristine** and have their fields initialised to **null**.

The rule D-UPDATE captures the semantics of an uncontended field update. The helper function **updateReturnType** calculates the subscript for the return value, based on the static types of the receiver and right-hand side value (*cf.*, E-UPDATE-*). Note that the receiver variable is nullified in the process, and the entire expression instead returns a new alias of the receiver with an updated type. This is a simple implementation of tracking how a variable changes types due to tentative writes (*cf.*, E-UPDATE-TENTATIVE).

Contended Operations (Figure 15). Because of the possibility of failure, contended operations are wrapped in conditionals, causing them to appear somewhat unwieldy. D-CAT-SUCCESS describes a successful **CAT** ($v_\epsilon = v_2$). The rule abstracts over the three possible shapes of **CAT** using the helper macro **C**, which returns a map ρ showing how variables' types are changed in the **then** branch, and an assignment map α of variables to be nullified. $\rho(e)$ denotes an expression with all substitutions in ρ performed. $\alpha(V)$ denotes a variable

$\boxed{cfg \hookrightarrow cfg'}$ *(Dynamic semantics)*

D-CAT-SUCCESS

$$\frac{\begin{array}{c} V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota) = (s, F) \qquad F(f) = v_\epsilon \qquad \langle H;\, V; p_1\rangle \overset{*}{\hookrightarrow} v_{1\,\mathtt{T}_1} \qquad v_\epsilon \;=\; v_1 \\ \langle H;\, V; p_2\rangle \overset{*}{\hookrightarrow} v_{2\,\mathtt{T}_2} \qquad \mathcal{F}\,(\mathtt{T}_x, f) = \mathit{mod}\, f : \mathtt{T}_f \qquad \mathbf{C}\,(\mathtt{T}_f, \mathtt{T}_2, (p_1, p_2)) = (\rho, \alpha) \end{array}}{\langle H;\, V; \mathbf{if}\,(\mathbf{CAT}\ (x.f, p_1, p_2))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{2\,\epsilon}])]; \alpha(V); \rho(e_1)\rangle}$$

D-CAT-RESIDUAL

$$\frac{\begin{array}{c} \langle H;\, V; \mathbf{if}\,(\mathbf{CAT}\ (x.f, p_1, p_2))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H';\, V';\, e_1'\rangle \\ V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota)(f) = v_\epsilon \qquad \langle H;\, V; p_1\rangle \overset{*}{\hookrightarrow} v_{1\,\mathtt{T}_1} \qquad v_\epsilon \;=\; v_1 \\ \langle H;\, V; p_2\rangle \overset{*}{\hookrightarrow} \iota_{\mathtt{T}} \qquad \mathcal{F}\,(\mathtt{T}, g) = \mathbf{var}\ g : \mathtt{T}_g \qquad H(\iota)(g) = v'_{\ \phi} \qquad z'\ \mathbf{fresh} \qquad e_1'' = e_1'[z_g \mapsto z'] \end{array}}{\langle H;\, V; \mathbf{if}\,(\mathbf{CAT}(x.f, p_1, p_2) \Rightarrow z_g)\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H';\, V', z' \mapsto v'_{\ \mathtt{T}_g};\, e_1''\rangle}$$

D-TRY-SUCCESS

$$\frac{V(x) = \iota_{\mathtt{T}_x} \qquad V(y) = v_{1\,\mathtt{T}_y} \qquad H(\iota) = (s, F) \qquad F(f) = v_{2\,\epsilon}}{\langle H;\, V; \mathbf{if}\,(\mathbf{try}\ (x.f = y))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{1\,*}])];\, V[y \mapsto \mathbf{null}_{\mathtt{T}_y}];^{x:\mathtt{T}_x \sim f}[e_1]\rangle}$$

D-FIX-SUCCESS

$$\frac{V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota) = (s, F) \qquad F(f) = v_{1\,\epsilon} \qquad V(y) = v_{2\,\mathtt{T}_y} \qquad v_{1\,\epsilon} \;=\; v_2}{\langle H;\, V; \mathbf{if}\,(\mathbf{fix}\ (x.f, y))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H[\iota \mapsto (s, F[f \mapsto v_{2\,*}])];\, V;^{x:\mathtt{T}_x \sim f}[e_1]\rangle}$$

D-CAT-FAIL

$$\frac{V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota)(f) = v_\phi \qquad \langle H;\, V; p_1\rangle \overset{*}{\hookrightarrow} v_{1\,\mathtt{T}_1} \qquad v_\phi \;\neq\; v_1}{\langle H;\, V; \mathbf{if}\,(\mathbf{CAT}\ (x.f, p_1, p_2))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H;\, V;\, e_2\rangle}$$

D-TRY-FAIL

$$\frac{V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota)(f) = v_*}{\langle H;\, V; \mathbf{if}\,(\mathbf{try}\ (x.f = y))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H;\, V;\, e_2\rangle}$$

D-FIX-FAIL

$$\frac{V(x) = \iota_{\mathtt{T}_x} \qquad H(\iota)(f) = v_{1\,\phi} \qquad V(y) = v_{2\,\mathtt{T}} \qquad v_{1\,\phi} \;\neq\; v_2}{\langle H;\, V; \mathbf{if}\,(\mathbf{fix}\ (x.f, y))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H;\, V;\, e_2\rangle}$$

D-STABLE-TRUE

$$\frac{V(x) = \iota_{\mathtt{T}} \qquad H(\iota)(f) = v_*}{\langle H;\, V; \mathbf{if}\,(\mathbf{isStable}\ (x.f))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H;\, V;^{x:\mathtt{T} \sim f}[e_1]\rangle}$$

D-STABLE-FALSE

$$\frac{V(x) = \iota_{\mathtt{T}} \qquad H(\iota)(f) = v_\epsilon}{\langle H;\, V; \mathbf{if}\,(\mathbf{isStable}\ (x.f))\,\{\, e_1 \,\}\ \mathbf{else}\ \{\, e_2 \,\}\,\rangle \hookrightarrow \langle H;\, V;\, e_2\rangle}$$

where the form of **CAT** is chosen by the shape of the arguments:

$$\begin{array}{rlcl} (\mathit{link}) & \mathbf{C}(\_, \mathtt{T}, (y.g, y)) & = & (\emptyset, \{y = \mathbf{null}_{\mathtt{T}}\}) \\ (\mathit{unlink}) & \mathbf{C}(\mathtt{T}, \_, (y, y.g)) & = & (\{y : \mathtt{T} \sim \mathtt{g}\}, \emptyset) \\ (\mathit{swap}) & \mathbf{C}(\mathtt{T}_f, \mathtt{T}_z, (y, z)) & = & (\{y : \mathtt{T}_f\}, \{z = \mathbf{null}_{\mathtt{T}_z}\}) \end{array}$$

Figure 15 Dynamic Semantics 2/2 (Contended operations). Note that $v_* \neq v'$ for all v and v'.

$\boxed{\Gamma \vdash cfg \quad \Gamma \vdash H \quad \Gamma; \mathtt{T} \vdash F \quad \Gamma \vdash T}$ *(Well-formed configuration)*

WF-CFG
$$\frac{\begin{array}{c}\Gamma \vdash H \qquad \Gamma \vdash V \qquad \Gamma \vdash T \\ \vdash \textbf{pristineness}\,(H, V, T) \\ \vdash \textbf{strongRestrictions}\,(H, V, T) \\ \vdash \textbf{linearOwnership}\,(H, V, T)\end{array}}{\Gamma \vdash \langle H; V; T\rangle}$$

WF-HEAP
$$\frac{\begin{array}{c}\forall\,\iota.\Gamma(\iota) = s \Rightarrow H(\iota) = (s, F) \\ \textbf{dom}\,(H) \subseteq \textbf{dom}\,(\Gamma) \\ \forall\,\iota.H(\iota) = (s, F) \Rightarrow \Gamma; s \vdash F\end{array}}{\Gamma \vdash H}$$

WF-VARS
$$\frac{\begin{array}{c}\forall\,x.\Gamma(x) = \mathtt{T} \Rightarrow V(x) = v_{\mathtt{T}} \\ \textbf{dom}\,(V) \subseteq \textbf{dom}\,(\Gamma)\end{array}}{\Gamma \vdash V}$$

WF-F-VAL
$$\frac{\begin{array}{c}\mathcal{F}\,(s, f) = mod\,f : \mathtt{T} \\ \Gamma \vdash v_{\mathtt{T}} : \mathtt{T} \qquad \Gamma; s \vdash F\end{array}}{\Gamma; s \vdash F, f \mapsto v_\phi}$$

WF-F-EMPTY
$$\frac{}{\Gamma; s \vdash \epsilon}$$

WF-T-E
$$\frac{\Gamma \vdash e : \mathtt{T}}{\Gamma \vdash e}$$

WF-T-FORK
$$\frac{\begin{array}{c}\Gamma = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e \qquad \Gamma_2 \vdash T\end{array}}{\Gamma \vdash e \parallel T}$$

$\boxed{\vdash \Gamma}$ *(Well-formed static typing environment)*

WF-ENV-E
$$\frac{}{\vdash \epsilon}$$

WF-ENV-X
$$\frac{\vdash \Gamma \quad x \notin \textbf{dom}\,(\Gamma) \quad \vdash \mathtt{T}}{\vdash \Gamma, x : \mathtt{T}}$$

WF-ENV-L
$$\frac{\vdash \Gamma \quad \iota \notin \textbf{dom}\,(\Gamma) \quad \vdash s}{\vdash \Gamma, \iota : s}$$

Figure 16 (Top) Well-formed configuration. Definitions of **pristineness** and **linearOwnership** can be found in Section 4. (Bottom) Typing environment.

map extended with the assignments of α.

The third argument of a **CAT** is nullified in linking or swapping **CAT**s. In the case of an unlinking or swapping **CAT**, the second argument of the **CAT** gets a new type corresponding to the respective static rules. D-CAT-RESIDUAL shows how additionally a residual alias can be introduced as a fresh variable z', as long as the underlying **CAT** succeeds. This rule uses direct substitution in the form of $e[z_g \mapsto z']$ rather than ${}^{x:\mathtt{T}}[e]$. This is because there is no change of types involved in residual aliasing.

$\langle H; V; p\rangle \overset{*}{\hookrightarrow} v_{\mathtt{T}}$ denotes the side-effect free evaluation of a stable path p. While we reduce the whole **CAT** in a single step, the type system rejects programs where the arguments p_1 and p_2 can change under foot—by B-CAT-* all paths are either local variables or stable **val** fields. Thus, the size of this atomic step is not important for the soundness or feasibility of our approach.

If the second argument of a **CAT** is not equal to the first, the write fails (D-CAT-FAIL). By definition, a fix pointer v_* is not equal to any value. (This is implemented by the `CAS` because fix pointers have their least significant bit set, which no aliases on the stack will).

Writing to a **once** field succeeds if the field is not yet fixed (D-TRY-SUCCESS). If the field is already fixed ($H(\iota)(f) = v_*$) the write fails (D-TRY-FAIL). Creating and installing a fix pointer is a contended write that attempts to update the existing value v of a field with v_* (D-FIX-SUCCESS). It atomically compares the current value stored in the field with the expected value, and if they are the same, updates the field with a fixed alias of the value. The operation fails if the expected value is not equal to the field value (D-CAT-FAIL).

D-STABLE-* checks whether a field contains a fix pointer or not.

4 Meta Theory

This sections describes the key properties of LOLCAT and sketches the proofs of why they hold. Full proofs can be found in the technical report [10]

Figure 16 defines well-formed configurations and environments. The definitions of a well-formed heap H and variable map V state that they are modelled by the environment Γ. In a well-formed heap, all objects have fields with values corresponding to their static type. A well-formed thread structure T consists of well-typed expressions. The "frame rule" $\Gamma = \Gamma_1 + \Gamma_2$ in WF-T-FORK makes sure that two parallel expressions cannot access the same local variables.

The rest of this section deals with three key definitions: **pristineness** states that if a reference has pristine type, then this reference has no aliases (this guarantees that such references are globally unique); **strongRestrictions** states that a strongly restricted field is globally inaccessible (*cf.,* the rely-guarantee interpretation in Section 2.1) and that two aliases cannot strongly restrict the same field; **linearOwnership** states that if two references are aliases that both allow reading the same **var** field, the static type of the paths to one of the references must prevent acquisition of the reference's ownership.

We prove the preservation of these properties separately, as a part of proving that well-formedness is preserved by the dynamic semantics. As a corollary of **linearOwnership** we show that reading or writing a **var** field is always free from data-races. To save space here, we summarize all the preservation properties with the following schema. The full definitions and proofs can be found in the technical report [10].

▶ Preservation of **X**. If a well-formed configuration $\langle H; V; T\rangle$ can take a step to $\langle H'; V'; T'\rangle$, this configuration will uphold **X**:

$$\Gamma \vdash \langle H; V; T\rangle \wedge \langle H; V; T\rangle \hookrightarrow \langle H'; V'; T'\rangle \Rightarrow\ \vdash \mathbf{X}$$

All three properties involve reasoning about the set of **references** in a configuration. A reference r ranges over fields $\iota.f$ in H, variables x in V, and locations ι appearing as subexpressions in T. Note that a reference is not an object but a means to access an object: after executing **let** x = y, x and y have the same value (are aliases), but are distinct references. If x is reduced to ι, this value, a (sub)expression in T, is also a distinct reference that aliases x and y.

The set of **movableReferences** is a subset of **references** and contains all r's whose value can be fully transferred into fields or variables of the same type. References in **val** and **once** fields are fixed, and therefore never in **movableReferences**. We explain the notation and the properties of references we are interested in below.

$H; V \vdash r : \texttt{T}$ means r has the (static) type $\texttt{T}$

$H; V \vdash r \hookrightarrow v$ means r has the value v

$H; V \vdash r$ **owns** $\iota.f$ means the value of r is ι and the field f is unrestricted in the type of r

$H; V \vdash r$ **reaches** $\iota.f$ means there is a path from r to $\iota.f$ that does not include restricted fields, and where all references in the path have types where f is unrestricted

$H; V \vdash r$ **reaches** $\iota.f$ **through** r' means r **reaches** a field $\iota.f$ through a path that ends with the reference r'

The intuition for the reachability properties is that if r **reaches** some field $\iota.f$, that field can be accessed through a series of operations on r. Note that LOLCAT does not allow the expression $x.f.g$, but requires $x.f$ to first be read into a local variable z. Thus, if $x.f$ **owns**

$\boxed{H;V \vdash r\ \textbf{owns}\ \iota.f \quad H;V \vdash r\ \textbf{reaches}\ \iota.f\ [\textbf{through}\ r']}$ *(see caption)*

REF-OWNS
$$\frac{H;\,V \vdash r \hookrightarrow \iota \qquad H;\,V \vdash r : \mathtt{T} \qquad \mathcal{F}(\mathtt{T}, f) = \mathit{mod}\, f : \mathtt{T}_f \qquad f \not\in \mathtt{T}}{H;\,V \vdash r\, \textbf{owns}\, \iota.f}$$

REF-REACHES-OWNS
$$\frac{H;\,V \vdash r\, \textbf{owns}\, \iota.f}{H;\,V \vdash r\, \textbf{reaches}\, \iota.f}$$

REF-REACHES-TRANSITIVE
$$\frac{H;\,V \vdash r : \mathtt{T} \qquad f \not\in \mathtt{T} \qquad H;\,V \vdash r' : \mathtt{T}' \qquad \vdash \mathtt{T}' \rightsquigarrow \mathtt{T} \qquad H;\,V \vdash r\, \textbf{reaches}\, r' \qquad H;\,V \vdash r'\, \textbf{reaches}\, \iota.f}{H;\,V \vdash r\, \textbf{reaches}\, \iota.f}$$

REF-REACHES-RESIDUAL
$$\frac{H;\,V \vdash r : \mathtt{T} \qquad \| f \in \mathtt{T} \qquad H;\,V \vdash \iota'.g : \mathtt{T}' \qquad \vdash \mathtt{T}' \rightsquigarrow \mathtt{T} \qquad H;\,V \vdash r\, \textbf{owns}\, \iota'.g \qquad H;\,V \vdash \iota'.g\, \textbf{reaches}\, \iota.f}{H;\,V \vdash r\, \textbf{reaches}\, \iota.f}$$

REF-REACHES-THROUGH
$$\frac{H;\,V \vdash r : \mathtt{T} \qquad f \not\in \mathtt{T} \qquad H;\,V \vdash r' : \mathtt{T}' \qquad \vdash \mathtt{T}' \rightsquigarrow \mathtt{T} \qquad H;\,V \vdash r\, \textbf{reaches}\, r' \qquad H;\,V \vdash r'\, \textbf{owns}\, \iota.f}{H;\,V \vdash r\, \textbf{reaches}\, \iota.f\, \textbf{through}\, r'}$$

Figure 17 Reference reachability.

g, we will require proper operations (*e.g.*, **CAT** or **consume**) to obtain z. How reachability changes after a successful pop in the Treiber Stack of Section 2.4 was illustrated in Figure 4.

Figure 17 defines reachability formally. Note that a reference with a strong restriction on a field f, and that **owns** some field through which it can reach another field $\iota.f$, also **reaches** $\iota.f$ (REF-REACHES RESIDUAL). This is because a series of **CAT**s ending with the extraction of a residual alias of $\iota.f$. REF-REACHES RESIDUAL instantiated for the Michael–Scott queue of Figure 6 states that `q.head` **reaches** `q.head.next.elem` because `elem` is strongly restricted in `q.head`'s type, and it **owns** `q.head.next`, which in turn **reaches** `q.head.next.elem`. On lines 40-41 of Figure 6, `q.head` is overwritten and a residual alias `elem` is introduced. Note that after this operation, `q.first` no longer **reaches** its own `elem` field.

Pristineness. The **pristineness** property states that if r is a reference of **pristine** type, there is no other reference r' that has the same value:

$$\begin{array}{l}
\vdash \textbf{pristineness}(H, V, T) \equiv \\
\quad \forall r \in \textbf{references}(H, V, T)\ . \\
\qquad H;V \vdash r : \textbf{pristine}\ \mathtt{t} \wedge H;V \vdash r \hookrightarrow \iota \Rightarrow \\
\qquad\quad \forall r' \in \textbf{references}(H, V, T)\ . \\
\qquad\qquad H;V \vdash r' \hookrightarrow \iota \Rightarrow r = r'
\end{array}$$

Proof sketch. We prove preservation of **pristineness** by induction over the structure of T. By assumption $\langle H; V; T\rangle$ is well-formed, and in particular all existing pristine references are globally unique. It is straightforward to show that reduction does not create any aliases of pristine references without nullifying that reference. ◀

Strong Restrictions. References with strongly restricted fields rely on these fields being globally inaccessible, which allows the creation of residual aliases. This precludes the existence of two aliasing references that strongly restrict the same field. If this was the case, they could both be used to extract the same residual alias, which would violate linear ownership.

To capture this, **strongRestrictions** states that if a reference r has the value ι and a type with a strongly restricted field f, $\iota.f$ is globally unreachable. Additionally, there can be no other reachable alias of r that also has f strongly restricted (**unreachable** is defined later):

$$
\begin{array}{l}
\vdash \textbf{strongRestrictions}(H, V, T) \equiv \\
\quad \forall r \in \textbf{references}(H, V, T) \ . \\
\quad\quad H; V \vdash r : \mathtt{T} \wedge \ \| f \in \mathtt{T} \ \wedge \ H; V \vdash r \hookrightarrow \iota \Rightarrow \\
\quad\quad\quad \textbf{unreachable}(\iota.f) \ \wedge \\
\quad\quad\quad \forall r' \in \textbf{references}(H, V, T) \ . \\
\quad\quad\quad\quad H; V \vdash r' : \mathtt{T}' \wedge \ \| f \in \mathtt{T}' \ \wedge \ H; V \vdash r' \hookrightarrow \iota \Rightarrow \\
\quad\quad\quad\quad\quad r = r' \vee H; V \vdash \textbf{unreachable}(r')
\end{array}
$$

This property exemplified for the Michael–Scott queue of Figure 6 states that the field `q.head.elem` must be globally unreachable at all times, and that there can be no aliases of `q.head` that also has `elem` strongly restricted. Note that **strongRestrictions** also precludes any other references into the queue with a strongly restricted `elem` field as this would violate the reachability provided through REF-REACHES-RESIDUAL (*cf.,* Figure 17).

Proof sketch. We prove preservation of **strongRestrictions** by induction over the structure of T. It is straightforward to show that a strongly restricted field f is always globally inaccessible since a value that flows into a reference where f is strongly restricted must have f unrestricted. The source of this value, which by **linearOwnership** held the sole ownership of f, is nullified or buried in the process. The same constraints on how values may flow into strongly restricted references make it straightforward to show that two such reachable references will never alias. ◀

Linear Ownership. Our relaxed notion of linearity allows unlimited aliasing, as long as ownership is linear. Operations maintains linearity by either transferring a pointer (**CAT** or consume), adding field restrictions on the source reference (possibly in combination with fix pointers), or by making the source reference inaccessible to the program (*cf.,* alias burying [4]). The latter is captured by **unreachable**:

$$
\begin{array}{l}
H; V; T \vdash \textbf{unreachable}(r) \equiv \\
\quad r = \iota.f \wedge \\
\quad \nexists r' \in \textbf{movableReferences}(H, V, T) \ . \ H; V \vdash r' \ \textbf{reaches} \ r
\end{array}
$$

After a tentative write, *e.g.,* Line 14 in Figure 3, the field written to will have overlapping ownership with some other reference. The tentative write changes the type of the target to one where the written field is transfer restricted, so the field is no longer considered reachable through the target. Since the target is a pristine value it has no aliases, and so the field is globally unreachable. Note that variables and locations are always considered reachable.

Sometimes an alias is reachable, but only from references whose types prevent them from transferring the ownership out of the reference. To reasoning about these situations we define a notion of *ownership burying*:

$$
\begin{array}{l}
H; V; T \vdash \textbf{buried}(r, \iota.f) \equiv \\
\quad H \vdash \textbf{stableField}(r) \ \wedge \\
\quad \nexists r' \in \textbf{movableReferences}(H, V, T) \ . \ H; V \vdash r' \ \textbf{reaches} \ \iota.f \ \textbf{through} \ r
\end{array}
$$

Let r be a reference whose value is ι. Now, r's ownership of the field $\iota.f$ is **buried** if r is some stable field $y.g$ (*i.e.,* is a **val** field or contains a fix pointer), and there are no movable references that can reach $\iota.f$ through it. Stability of $y.g$ ensures that we cannot **CAT** against $y.g$ to acquire the value in $\iota.f$ (for example by unlinking ι: $z = y.g; \textbf{CAT}(y.g, z, z.g')$).

The condition that no references reach $\iota.f$ through $y.g$ ensures that even if there is some speculatable field $x.f'$ aliasing y ($x.f'$ **reaches** $y.g$ which in turn **owns** $\iota.f$), and we swing $x.f'$ forward to alias $y.g$ by $\mathbf{CAT}(x.f', y, y.g)$, the type of $x.f'$ does not grant access to f. This was exemplified in Figure 4: after the pop, the field `a.next` still owns `b.elem`, but this ownership is buried (and therefore benign).

Finally, **linearOwnership** states that if two different references in a configuration alias some location ι, and can read or write the same **var** field $\iota.f$ (they both **own** $\iota.f$), then either (1) one of the references' ownership of $\iota.f$ is **buried**, or (2) one of the references is **unreachable**:

$$\begin{array}{ll}
\vdash \mathbf{linearOwnership}(H, V, T) \equiv & \\
\quad \forall \iota, f\ .\ H \vdash \mathbf{varField}(\iota.f) \Rightarrow & \\
\quad\quad \forall r_1, r_2 \in \mathbf{references}(H, V, T)\ . & \\
\quad\quad\quad H; V \vdash r_1 \ \mathbf{owns}\ \iota.f \wedge H; V \vdash r_2 \ \mathbf{owns}\ \iota.f \Rightarrow & \\
\quad\quad\quad\quad r_1 = r_2 & \\
\quad\quad\quad\quad \vee H; V; T \vdash \mathbf{buried}(r_1, \iota.f) \vee H; V; T \vdash \mathbf{buried}(r_2, \iota.f) & (1) \\
\quad\quad\quad\quad \vee H; V; T \vdash \mathbf{unreachable}(r_1) \vee H; V; T \vdash \mathbf{unreachable}(r_2) & (2)
\end{array}$$

Note that a reference $\iota'.g$ being unreachable still allows aliases of ι', but any such alias must have g transfer restricted, meaning it appears as a **val** field and cannot be acquired by a **CAT**.

Proof sketch. We prove preservation of **linearOwnership** by induction over the structure of T, making sure that whenever an alias owning some **var** field is introduced, (1) and (2) are preserved. The proof also shows that no configuration changes affect reachability from existing references in such a way that (1) or (2) is violated.

In any well-formed configuration we have a set of references $\overline{r}$ (fields, variables and free locations) for which **linearOwnership** holds. An observation we make use of in the proof is that if we step to a configuration where the new set of references $\overline{r'}$ is a subset of $\overline{r}$ and any types that might have changed are more restricted than the original types we do not break (1) or (2), since these are ultimately concerned with which references are *not* reachable. The intuition here is that removing a reference cannot make a previously unreachable reference reachable. Similarly, restricting fields in the type of a value will not enable reaching any references previously unreachable since restrictions shrink the set of reachable fields. ◀

Corollary: Data-Race Free Var Field Accesses. A corollary of **linearOwnership** is that two variables on the stack can never alias unless the intersection of their accessible **var** fields is empty. This means that reading or writing a **var** field $x.f$ is always free from data-races, as there can never be a variable y aliasing x that can read or write the same field.

5 Prototype Implementation & OO Support

We have a prototype implementation of LOLCAT in a fork of the Encore programming language [9]. Encore is an actor-based object-oriented programming language with trait-based inheritance and a capability-based type system that includes a **linear** capability denoting the only reference to an object in the system, and a **subordinate** capability denoting a reference that may not escape its enclosing structure (key to providing actor isolation).

We extend Encore with a **lockfree** capability, **once** fields, **spec** fields, and associated operations. Speculative reads are explicated using a **speculate** keyword. An object with a

lockfree capability must not contain **var** fields, and may thus be aliased freely. We restrict **once** and **spec** fields to hold values whose types are *both* **linear** and **subordinate**, and relax the semantics of the **linear** capability to allow non-destructive reads following the LOLCAT rules. As subordinate objects may not escape their enclosing objects, this restricts the data-flow of linear references and encapsulates all shared mutable state inside **lockfree** capabilities. This is useful for garbage collection (*cf.* Section 5.3) and keeps LOLCAT specific types isolated.

We have used our prototype to implement the examples of this paper (using loops rather than recursion). Additionally we have implemented a dictionary based on Fomitchev and Ruppert's lock-free skip list [19], as well as a set based on the lock-free binary search tree by Ellen *et al.* [16]. LOLCAT can also be used to implement simpler constructs such as spin-locks, a variant of which has been used to implement the lazy list-based set by Heller *et al.* [29].

5.1 Implementing CAT and Fix Pointers

The Encore compiler is a source-to-source translator from Encore into C11. The **spec** fields and **once** fields are implemented as word-aligned fields in structs that correspond to classes. Fix pointers are implemented using a mark bit in the least significant bit of pointer addresses. Consequently, reading speculative fields and once fields involve masking this bit out which causes some overhead.

Well-typed linking and unlinking **CAT**s desugar into several statements surrounding a swapping **CAT**. The statement **if** **CAT**(x.f, y, y.g) **then** e_1 **else** e_2, desugars to:

```
let tmp = speculate y.g in // tmp fresh
  if CAT(x.f, y, tmp) then e1 else e2
```

The swapping **CAT** is translated into C as `CAS(&x.f, y, tmp)`. The **try** and **fix** expressions are translated similarly, but also manipulate the mark bit; *e.g.,* **try**(x.f, y) desugars into:

```
void *z = mark_least_significant_bit(y); CAS(&x.f, y, z);
```

Future work involves support for installing fix pointers in multiple fields of the same object atomically through a double-word `CAS` in the case of two (adjacent) fields, and a hidden pointer indirection to an immutable tuple of pointers in the case of more than two fields.

5.2 LOLCAT and Object-Oriented Programming

Extending LOLCAT with support for object-oriented programming is straightforward. The key problem going from procedural to object-oriented is solving the issue of self typing in the presence of field restrictions. In a procedural setting, changing the type of a variable x from `T` to `T | f` propagates the restriction on `f` because x must be passed to all functions as an explicit argument, requiring that the function's signature has a type which is at least as restrictive. With object-oriented programming, care must be taken so that the restriction does not only apply to the *client view* of the object but to the object's *view of itself.* Several approaches exist in the literature: annotate each method to reflect the self type (*e.g.,* [39, 50]); employ an effect system [47] that captures what variables are read and disallow $x.m()$ on x : `T | f` if m reads or writes x (*e.g.,* [24, 11]); or use program analysis.

While Encore does not have an effect system as such, traits in Encore explicitly *require* fields and *provide* methods. A trait's methods can only use fields it requires. This enables straightforward field restrictions: if x : `T | f`, then $x.m()$ is allowed only if m is defined in a trait that does not require `f`. This admittedly somewhat coarse-grained support for restriction is still enough to implement all examples mentioned above.

5.3 Garbage Collection and ABA

Traditional linear types have been useful in the past to detect when a value can be deallocated without causing dangling pointers. Our relaxation of linearity notably excludes this use. In this paper we have implicitly assumed garbage collection (GC), and Encore is also a garbage collected language. However, the invariants of LOLCAT can be made to hold without GC. Without a GC, we could automatically compile **CAT**s to use a monotonically increasing counter (*cf.,* [33]) in combination with pointer identity, effectively implementing an LL/SC on-top of CAS, to avoid ABA problems.

Recently, in the context of the implementation of LOLCAT in Encore, Yang and Wrigstad devised Isolde [53], a slot-in GC protocol that leverages the LOLCAT type system. Isolde manages the memory in each lock-free data structure separately from the rest of the system, and does not stop threads from making progress for GC. Notably, Isolde relies on identifying the type of **CAT** to insert different GC behaviour.

6 Related Work

We are not aware of other type systems aimed at implementing lock-free algorithms, or type systems that allow atomic transfer of ownership without using locks or destructive reads. Specifically, we have not seen types that give meaningful static semantics to the `CAS` primitive. There are several type systems for programming with linear (or unique) references, alone *e.g.,* [32, 39, 18, 4] or with other techniques [7, 1, 12, 3, 6, 8, 42, 13, 25, 22], or systems with linear reference permissions [52, 45]. These systems rely on one or more of the following techniques:

1. Destructive reads enforce strict linearity;
2. Alias burying allows aliases of linear variables guaranteed to be updated before next use;
3. Borrowing allows temporary violations of linearity for a well-defined scope, after which linearity is re-established;
4. Linear references are guaranteed to be the only reference to an object *outside of the object's representation.*

Several of the systems above use linearity in the context of concurrent and parallel programming to avoid data-races, *e.g.,* holding the only reference to an object guarantees absence of concurrent readers or writers. None of the systems handle lock-free programming because of the reliance of destroying or burying aliases. Conceptually, the "life cycle" of a linear reference in LOLCAT: newborn $\rightarrow$ published $\rightarrow$ acquired is similar to borrowing, but we are statically never able to get back to a strictly linear state after publication. Similarly, acquisition is also conceptually similar to burying: we transfer an owning reference to the stack, where it is guaranteed to remain until the current thread voluntarily gives it up.

Wadler notes that linear values can be deallocated as soon as they are used [51]. Kobayashi's Quasi-Linear types allow deallocating a linear value when it goes out of scope [35]. Our relaxation of linearity prevents this optimisation as linear objects may have restricted aliases. Ennals *et al.* define a concurrent linearly typed programming language for packet processing which relaxes linearity to allow multiple references *from the same thread* [17].

Turon defines reagents, basic building blocks for lock-free programming [49]. These simplify implementation of lock-free data structures, but do not provide any guarantees of data-race freedom for acquired references.

Militão uses Rely-Guarantee Protocols to guarantee safe interference over shared memory, allowing unbounded aliasing but only a single linear capability to an object [37]. The transfer of this capability is very similar to LOLCAT's ownership transfer, but uses locks to ensure mutual exclusion, whereas LOLCAT allows non-blocking atomic transfer of ownership.

Gordon uses Rely-Guarantee References to verify functional correctness of lock-free data structures [21]. This system is more expressive than ours, but also more heavyweight as it requires writing specifications and mechanised proofs. Gotsman *et al.* use rely-guarantee reasoning for similar purposes but also develop a tool for automating the proofs [23]. Haziza *et al.* develop a tool that can automatically verify correctness of the Treiber Stack and Michael–Scott Queue with little or no hints from the programmer [28].

Compared to systems that facilitate manual or automatic verification of lock-free algorithms, LOLCAT trades reasoning power for simplicity and modularity. LOLCAT types have meaning on their own and provide useful invariants without requiring inter-procedural analysis; looking at the types of a piece of code explains how it this code affects ownership.

7 Conclusions & Future Work

LOLCAT is a type system for lock-free programming with linear types. It provides static semantics for a number of patterns found in lock-free programming, such as speculation, publication and acquisition. Specifically, it gives meaningful types to the `CAS` primitive which precisely describe ownership transfer in linked data structures. The type system is expressive enough to encode several algorithms from the literature on lock-free programming.

In future work, we will develop a library of lock-free data structures in our Encore implementation to further evaluate the expressiveness of LOLCAT. We are currently working on a garbage collection scheme that uses our types to achieve pause-free garbage collection [53]. We will also consider other correctness aspects of lock-free algorithms and investigate how the language can be extended to further aid programmers in writing correct lock-free code.

Acknowledgments. We are grateful for the comments from Dave Clarke, Sophia Drossopoulou, the SLURP reading group at Imperial College, and the anonymous reviewers.

References

1 Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *ACM SIGPLAN Notices*, volume 37, pages 311–330. ACM, 2002.

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

3 Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *ACM SIGPLAN Notices*, volume 36, pages 56–69. ACM, 2001.

4 John Boyland. Alias burying: Unique variables without destructive reads. *Softw., Pract. Exper.*, 31(6):533–553, 2001.

5 John Boyland. The interdependence of effects and uniqueness. In *Workshop on Formal Techs. for Java Programs*, 2001.

6 John Boyland. Checking interference with fractional permissions. In *Static Analysis*, pages 55–72. Springer, 2003.

7 John Boyland, James Noble, and William Retert. Capabilities for sharing. In *ECOOP 2001—Object-Oriented Programming*, pages 2–27. Springer, 2001.

8 John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. *ACM SIGPLAN Notices*, 40(1):283–295, 2005.

9 Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, EinarBroch Johnsen, KaI. Pun, S.LizethTapia Tarifa, Tobias Wrigstad, and AlbertMingkun Yang. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 1–56. Springer International Publishing, 2015. `doi:10.1007/978-3-319-18941-3_1`.

10 E. Castegren and T. Wrigstad. Lolcat: Relaxed linear references for lock-free programming. Technical Report 2016-013, 2016. Uppsala University. URL: `http://www.it.uu.se/research/publications/reports/2016-013/`.

11 Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN Notices*, volume 37, pages 292–310. ACM, 2002.

12 Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. *ECOOP 2003*, pages 59–67, 2003.

13 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. *Programming Languages and Systems*, pages 139–154, 2008.

14 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *AGERE*, 2015.

15 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *ECOOP 2001—Object-Oriented Programming*, pages 130–149. Springer, 2001.

16 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010.

17 Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Programming Languages and Systems*, pages 204–218. Springer, 2004.

18 Manuel Fahndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM SIGPLAN Notices*, volume 37, pages 13–24. ACM, 2002.

19 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.

20 Anwar Ghuloum. Face the inevitable, embrace parallelism. *Communications of the ACM*, 52(9):36–38, 2009.

21 Colin S Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

22 Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 21–40. ACM, 2012.

23 Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *ACM SIGPLAN Notices*, volume 44, pages 16–28. ACM, 2009.

24 Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99—Object-Oriented Programming*, pages 205–229. Springer, 1999.

25 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP 2010–Object-Oriented Programming*, pages 354–378. Springer, 2010.

26 Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

27 Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 1, pages 300–314. Springer, 2001.

28 Frédéric Haziza, Lukáš Holík, Roland Meyer, and Sebastian Wolff. *Pointer Race Freedom*, pages 393–412. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. `doi:10.1007/978-3-662-49122-5_19`.

29 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems*, pages 3–16. Springer, 2005.

30 Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *ACM SIGPLAN Notices*, volume 25, pages 197–206. ACM, 1990.

31 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

32 John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM SIGPLAN Notices*, volume 26, pages 271–285. ACM, 1991.

33 IBM. Ibm system/370 extended architecture, principles of operation, 1983. publication no. SA22-7085.

34 Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

35 Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 29–42. ACM, 1999.

36 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*, pages 267–275. ACM, 1996.

37 Filipe Militão. *Rely-Guarantee Protocols for Safe Interference over Shared Memory*. PhD thesis, Carnegie Mellon University & Universidade de Lisboa, 2015.

38 Filipe Militão, Jonathan Aldrich, and Luís Caires. Aliasing control with view-based typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, page 7. ACM, 2010.

39 Naftaly H Minsky. Towards alias-free pointers. In *ECOOP'96—Object-Oriented Programming*, pages 189–209. Springer, 1996.

40 Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

41 Peter Müller. *Modular Specification and Verification of Object-oriented Programs*. Springer-Verlag, Berlin, Heidelberg, 2002.

42 Peter Müller and Arsenii Rudich. Ownership transfer in universe types. In *ACM SIGPLAN Notices*, volume 42, pages 461–478. ACM, 2007.

43 Johan Östlund. *Language Constructs for Safe Parallel Programming on Multi-cores*. PhD thesis, Department of Information Technology, Uppsala University, Jan 2016.

44 Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer, 1985.

45 Francois Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

46 Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

47 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

48 R Kent Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

49 Aaron Turon. Reagents: expressing and composing fine-grained concurrency. In *ACM SIGPLAN Notices*, volume 47, pages 157–168. ACM, 2012.

50 Jan Vitek and Boris Bokowski. Confined types in java. *Software: Practice and Experience*, 31(6):507–532, 2001.

51 Philip Wadler. Linear types can change the world. In *IFIP TC*, volume 2, pages 347–359. Citeseer, 1990.

52 Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical permissions for race-free parallelism. In James Noble, editor, *ECOOP 2012*, volume 7313 of *LNCS*, pages 614–639. Springer, 2012. `doi:10.1007/978-3-642-31057-7_27`.

53 Albert Mingkun Yang and Tobias Wrigstad. Type-assisted automatic garbage collection for lock-free data structures. In *International Symposium on Memory Management*, 2017.

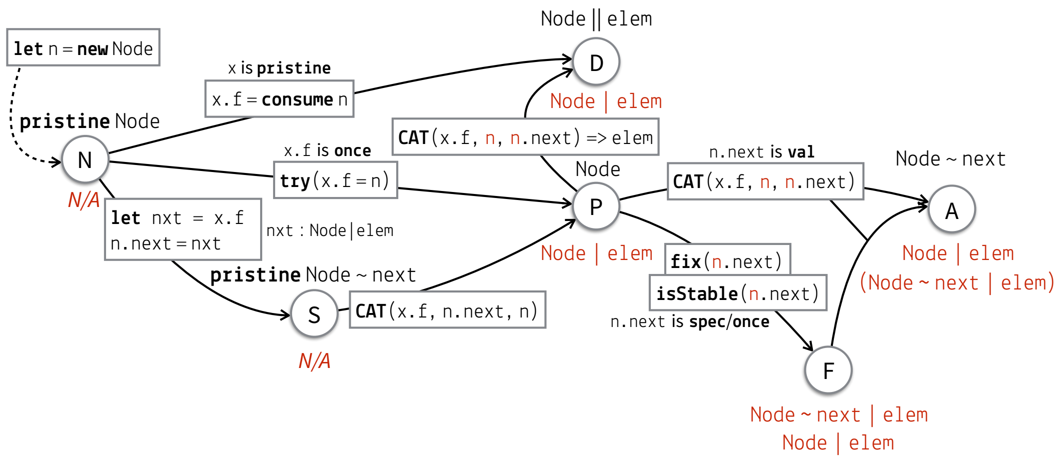
A Type Transitions in Lolcat

The figure below shows how the type of a node in a linked data structure changes to reflect the different views of the node during its lifetime. It is an extended version of Figure 1b with added transition labels showing which operations changes the view of the node. Depending on the data structure and operation at hand, the field `x.f` represents the `top` field of a stack, the `first` field of a queue, or the `next` field of another node. Other transitions are possible, but we focus on operations that appear in our examples and where ownership changes.

The types above each view is the type of the node as seen by its owning reference, which may be stored in a local variable on the stack, or in a field on the heap. The types below each view (in red) show which types aliases of the node may have. In the transition labels, when the variable `n` is red it is a speculative alias of the node.

The labels on the nodes refer to the same views as in Figure 1a: **N**ewborn, **S**taged, **P**ublished, and **A**cquired. It also includes the view **D**ummy, where ownership of a **var** field has been permanently buried, and the view **F**ixed, where a **spec** or **once** field has been made immutable. In the Fixed view, there is no single type that completely describes the node; the internal type `Node` does not show that the `next` field has been fixed, but the external type `Node`∼`next` | `elem` does not have ownership of the `elem` field. Once a fixed node has been acquired however, the acquiring thread again sees the fully accurate type `Node`∼`next`. If the node was fixed before being acquired, aliases of type `Node`∼`next` | `elem` may still exist, as well as "normal" speculative aliases of type `Node` | `elem`.

The transition "Newborn → Dummy" permanently forgets the ownership of the `elem` field. The transition "Published → Dummy" allows extracting a residual alias `elem` since the owning reference in `n.next` is buried in a strongly restricted field (*cf.* Section 2.5). Note that in this transition, it is actually the view of the node originally in `n.next` which changes.


Node || elem
let n = new Node
x is pristine
x.f = consume n
D
Node | elem
pristine Node
CAT(x.f, n, n.next) => elem
N
x.f is once
try(x.f = n)
Node
n.next is val
CAT(x.f, n, n.next)
Node ~ next
P
A
N/A
let nxt = x.f
n.next = nxt
nxt : Node | elem
Node | elem
fix(n.next)
Node | elem
(Node ~ next | elem)
pristine Node ~ next
isStable(n.next)
S
CAT(x.f, n.next, n)
n.next is spec/once
F
N/A
Node ~ next | elem
Node | elem