Capable: Capabilities for Scalability

Elias Castegren and Tobias Wrigstad

Uppsala University

Abstract This paper describes the current state of the design of the Capable system, which explores the use of capability-based alias management for parallel programming. Capable allows aliasing as long as all aliases offer non-conflicting capabilities to possibly shared resources, thereby avoiding the need to propagate effect information and opening up for flexible programming which is generally not possible with effect systems. Classes are formed from combinations of capabilities, and combinators influence object layout to avoid false sharing. This allows reusing a single declaration in both memory-efficient and parallel-efficient ways. Finally, Capable capabilities supports patterns of lock-free programming with the goal of providing a limited form of static checking for typical lock-free idioms.

1 Introduction and Background

The need for alias management in stateful programs has been greatly exacerbated since the rise of multicore. Recent years have seen a wealth of proposals for restricting and controlling aliases in the presence of (increasingly ubiquitous) parallelism. In this paper, we report on the design of Capable—a capability-based static type system for parallel programming. The goal is to support common parallel programming patterns with a minimum of annotation overhead, and without imposing a structuring principle on how programs are written.

The two most common approaches to alias management are *prevention* and *control*, using the terminology from the Geneva Convention on the Treatment of Object Aliasing [10]. Alias prevention schemes impose restrictions on how aliases may be created, such as uniqueness or ownership, to exclude undesirable situations by design. Alias control schemes instead freely allows aliasing, but exclude undesirable situations by restricting their use, for example using effect systems or similar techniques such as read-only references. It is common for systems to combine both these approaches.

Unique pointers are very powerful and give useful guarantees for a wealth of programming situations, but exclude many common programming patterns making it inherently complicated to program with uniqueness. Despite being a lot less restrictive, ownership models such as owners-as-dominators and owners-as-modifiers [6] also exclude common patterns, and programmers must often choose between less restrictive programming at the price of too coarse-grained control, or fine-grained control at the price of too restrictive programming.

Effect systems offer additional degrees of freedom. This is nicely illustrated in the Mojo system [4] which does not enforce an owners-as-dominators *structure* on a program, but excludes use of pointers outside owners-as-dominators by way of an effect system. Effect systems are powerful and flexible, but also problematic. Method annotations are a standard way to advertise computational effects to exclude dangerous races. An effect system can guarantee the absence of deadlocks, but has the consequence that long-running methods which conditionally and/or briefly touches some resource effectively locks this resource for the entire duration of the method call. Furthermore, effect annotation overhead can be both restrictive on polymorphism, and incur a hefty syntactic baggage on deep call-chains. Some systems, for example ownership-based effect systems, allow reducing syntactic baggage at the cost of more coarse-grained effect footprints of operations.

Capable takes the alias prevention approach to creating aliases to objects. In Capable, all aliases can be used freely in parallel, and alias control techniques are applied to govern their creation rather than use. This avoids effect annotations and gives rise to a simple system which is able to handle situations where aliases are created and destroyed dynamically, even in a non-structured fashion, while being statically checkable.

Capable does not attempt to control aliasing to perform arbitrary verification tasks. Instead, Capable takes the view that aliasing plus parallelism is "dangerous". Objects are divided into two groups: objects that may be freely aliased across threads, and objects that may not. This dichotomy is accompanied by a system that enforces (non-)sharing laws and synchronisation (or similar) for objects which may be shared across threads. The result of this pragmatic setup is simplification. The current design of Capable is as a traits-based programming language with garbage collection and static, manifest typing. To operate on an object in parallel, the object must either be thread-safe (we will return to this definition later), or it must be possible to (logically) carve the object up into isolated parts which can be modified in parallel without causing interference.

Outline $ 2–2.3 describes the Capable fundamentals, capabilities, traits, and composition. $ 2.4 connects composition to object layout. $ 2.5–2.7 discusses aliasing and capabilities. $ 3 details how Capable capabilities can be married to lock-free programming. $ 4 discusses related work. $ 5 concludes.

2 Capable

In Capable, references are synonymous with capabilities and objects with resources. Capable capabilities are divided into two categories: *exclusive* and *non-exclusive*, denoted c and c^* below. Exclusive capabilities must be treated linearly in the program, and are therefore safe from low-level races[1]—any resource to which an exclusive capability governs access cannot be accessed in parallel. Non-exclusive capabilities can further be divided into *safe* and *unsafe* capabilities, where a safe

[1] High-level races are still possible, but requires that the program explicitly transfers an exclusive capability back and forth between threads.

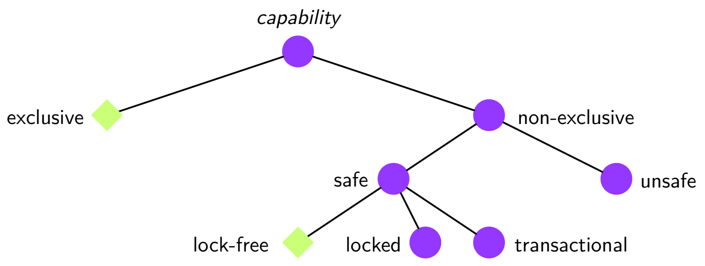


Figure 1. Partial Capable capability hierarchy. The highlighted squares show our main focus for this presentation although remaining bits are not without merit.

capability is thread-safe by construction and unsafe capabilities are not. There are two ways of approaching unsafe capabilities, depending on the desired semantics. Either one simply accepts that use of unsafe capabilities are subject to races and other non-deterministic behaviour, or one requires that they are sufficiently "protected" at use-site, *e.g.,* by locks or transactions, to exclude such behaviour. For brevity, we exclude unsafe capabilities from this treatise, although they have important merits. The means of achieving thread-safety is largely orthogonal to the core Capable design. For example, different subcategories of safe capabilities exist for different approaches such as *locked* (using an intrinsic lock acquired and released by all methods similar to Java synchronisation) and *transactional* (using STM), etc. Figure 1 shows the partial Capable capability hierarchy.

The term thread-safe warrants further discussion. An object with only synchronised methods can be considered thread-safe on a low level, meaning that no two threads can manipulate it's innards at the same time. However, interactions which span more than a single method can still be interleaved which could violate thread-safety on a higher level. In this paper, we are mostly concerned with the low-level notion of thread-safety.

2.1 Capabilities, Traits, Classes and Subtyping

In Capable, capabilities are introduced into a program using traits [7]. For now, it suffices to think of a trait t as a set of provided services which we denote $[\![t]\!]$ (these are operations allowed on the trait, *i.e.,* the method definitions in the trait declaration) defined in terms of a set of required services $R(t)$ (the instance variables, which may be writable, readable or both). Each trait t introduces a new capability c_t. Whether a trait introduces an exclusive, safe or unsafe capability is declared in the trait header. For brevity, we omit the discussion on how to enforce or statically check that a trait's implementation conforms to its capability kind. The specific case of *safe, lock-free* traits is the topic of Section 3.1.

Classes are formed by composing traits, making a class type a *composite capability.* A class is a set of instance variable declarations V and a set of traits $\{t_1, \ldots, t_n\}$ such that $V = \bigcup_{i=1..n} R(t_i)$. A class provides the sum of the services of the traits it is composed of. Composition of capabilities is detailed further in

Section 2.2. For simplicity, traits that make up a class may not have conflicting names of provided services, but may overlap in required services. Two or more traits can share a single instance variable. For example, two traits in a single class can share a variable which may be read and written by both; a channel between two traits is easy to encode by giving one read rights and one write-rights.

Subtyping is defined using set inclusion, where elements of the set are nominal capabilities, so c is a subtype of c' if $c = c'$ or c is a composite capability that includes c' (directly or transitively).

A notion of "capability subsumption" arises naturally as a side-effect of storing one exclusive capability inside a field of another capability (of any kind). For example, let c_1 be a capability which holds a reference with an exclusive capability c_2. In this case, c_1 clearly subsumes c_2 as there is no way to interact with c_2 except through the interface of c_1. (Of course, it is also possible to transfer c_2 out of c_1, in which case the subsumption ceases.) If c_2 is a non-exclusive capability, c_2 may be referenced from elsewhere in the program. Still, interaction with c_2 in c_1 is unconstrained, modulo enforced uses of proper guards—if c_2 is safe then all parallel interaction is also safe, if c_2 is unsafe we either permit unsafe behaviour or were forced to properly guard all uses of c_2. In all these cases, there is never a need to advertise the existence of c_2 or track side-effects on c_2 in c_1's interface. This is a design goal of Capable: to avoid (the equivalence of effect) annotations of object use which must be propagated through the program, and impacts negatively on refactoring, polymorphism and reuse.

Traits may be parameterised over types to form nested capabilities (see below). In addition to standard reuse, this mechanism is used to *expose* capability subsumption to allow additional parallelism. Bounds on type parameters control how the trait may use objects of the parameter type internally, for example by restricting the parameter to be *safe*, or a subtype of some other type.

2.2 Capability Composition

Capabilities are formed by composition of one or more capabilities. There are three composition operators: conjunction, $\otimes$; disjunction, $\oplus$; and nesting, $\langle\ \rangle$.

Conjunction and Disjunction A conjunctive composite capability $c_1 \otimes c_2$ supports the same operations as its subcapabilities and may be *split* into two separate capabilities c_1 and c_2, which may be used in parallel with no need for mutual coordination. Two capabilities sharing a common instance variable may only be used in a conjunction if there is no write-write conflict on the variable, *i.e.,* if $v \in R(c_1 \otimes c_2)$ and v is writable by c_1 then v must not be writable in c_2. Note that conjunction permits racy read–write behaviour.

A disjunctive composite capability $c_1 \oplus c_2$ also supports the same operations as its subcapabilities, however splitting a disjunction yields *either* c_1 or c_2, never both. There are no restrictions on the shared instance variables of two capabilities used in a disjunction.

Suppose we have a class C composed as $t_1 \oplus t_2 \otimes t_3$. A reference of type C is a composite capability $c_1 \oplus c_2 \otimes c_3$, where c_i is the capability introduced by trait

t_i. This capability can be split either into the capability c_1 or into the composite capability $c_2 \otimes c_3$. The latter can be further split into two capabilities c_2 and c_3. This captures that it is unsafe to use c_1 in parallel with c_2 or c_3.

As a concrete example of composite capabilities we can build a capability for a `Cell` such that $\texttt{Cell} = \texttt{set} \oplus \texttt{get}^*$ and a `Pair` such that $c_2 = \texttt{Cell} \otimes \texttt{Cell}$.[2] We imagine that `set` and $\texttt{get}^*$ are traits for setting and getting a stored value. The disjunctive composition of these traits expresses that these operations are not safe to perform in parallel. A `Cell` may be split into either a `set` capability or a $\texttt{get}^*$ capability, but never both. Note that $\texttt{get}^*$ is non-exclusive (and safe) and can therefore be aliased freely after the split. The `Pair` capability is similar to the `Cell`, but allows a pointer to a pair to be split into *two* pointers which can be operated on (`get` and `set`) in parallel. Note that these two new pointers are aliases of the same `Pair` as they were split from — we only restrict the operations allowed through them.

Nesting Nested capabilities are introduced by the parametric traits briefly mentioned above. If c_1 is defined to take a parameter P, then the operations of the nested composite capability $c_1\langle c_2\rangle$ where P is instantiated by c_2 are such that $[\![c_1\langle c_2\rangle]\!] = [\![c_1]\!]\{^{c_2}/_P\}$, where the substitution means that all occurrences of the capability parameter P in the operations of c_1 are replaced by c_2.

If c_1 is a non-exclusive capability (so that multiple copies of it can be created) and $c_2 = c \otimes c'$, then splitting a nested capability $c_1\langle c_2\rangle$ produces two nested capabilities $c_1\langle c\rangle$ and $c_1\langle c'\rangle$. If $c_2 = c \oplus c'$, then splitting produces either of the two, but not both.[3] Note the requirement that c_1 be non-exclusive, as otherwise the splitting would have broken exclusivity by introducing copies of exclusive capabilities to the same resource. If c_2 were a non-exclusive capability and $c_1 = c \otimes c'$, splitting would produce the nested capabilities $c\langle c_2\rangle$ and $c'\langle c_2\rangle$ etc.

To exemplify nested capabilities, imagine a capability for a `List` of c's such that $\texttt{List}\langle c\rangle = \texttt{add}\langle c\rangle \oplus \texttt{del}\langle c\rangle \oplus \texttt{ith}^*\langle c\rangle$. The `List` capability is a combination of smaller capabilities for adding, deleting and getting elements from the list. This means that a pointer to a list can be specialised into a pointer with either of these three capabilities, and in case of `ith` into multiple pointers with just an `ith`-capability. If c is a composite capability $c_1 \otimes c_2$, a capability of type $\texttt{ith}^*\langle c\rangle$ can be turned into two (aliasing) capabilities of type $\texttt{ith}^*\langle c_1\rangle$ and $\texttt{ith}^*\langle c_2\rangle$.

2.3 Putting it all Together 1: A Simple Cell

Figure 2 shows how a cell class can be formed by disjunctive composition of two capabilities, one which needs exclusive access to the object (set) and one that is thread-safe in itself as it only performs reads.

[2] In the definition of the `Pair` class, and when splitting a `Pair`, there needs to be some disambiguation to distinguish between the two `Cell`s. We omit this for brevity.

[3] If there is a bound c_3 on the parameter P to c_1, to which c_2 is bound, then $c <: c_3$ and $c' <: c_3$ must hold.

```
safe trait Get<safe T> {
  require T value;
  T get() { return value; }
}

class Cell<safe T> = Get<T> + Set<T> {
  T value;
}
```

```
exclusive trait Set<T> {
  require T value;
  void set(T v) {
    value = consume v;
  }
}
```

Figure 2. A Cell is composed from Get and Set traits which allow reading and updating a common field. To keep things simple for this presentation, we require names of declared fields of classes to match required fields in traits.

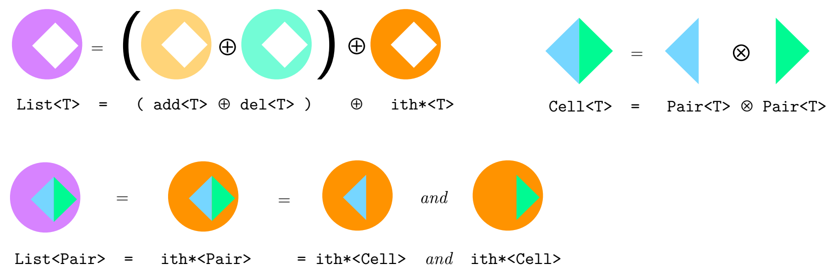
2.4 Capability Composition and Parallel Programming

Modulo the unguarded use of unsafe non-exclusive capabilities, Capable programs only contain aliases (capabilities) which are safe to use in parallel. We have used splitting as a key technology to operate on a single object in parallel by allowing the creation of multiple aliases to (possibly different parts of) an object. Splitting is a mostly logical operation and aliases created by splitting are identical pointers.

Nested composites may present interesting opportunities for parallel programming. The capability $\texttt{List}\langle\texttt{Pair}\rangle$ using the definitions above can be turned into $\texttt{ith}^*\langle\texttt{Pair}\rangle$, which in turn can be split into two $\texttt{ith}^*\langle\texttt{Cell}\rangle$ which accesses the first and second compartment of the pair respectively. The two capabilities correctly capture the safe "iteration" (here ineffectively expressed using multiple calls to `ith`) over the stable list to perform safe parallel mutations on elements in the list. Figure 3 shows this process pictorially.

Since $c_1 \otimes c_2$ and $c_1 \oplus c_2$ have different splitting semantics, composition also gives hints to how resources should be *laid out in memory* with respect to cache behaviour. In the former case, the combined resource may be operated on in parallel, so to avoid false sharing the fields in c_1 and the fields in c_2 should never be placed on a single cache line. In the latter case, the combined resource is never used in parallel, which allows more space-efficient placement of fields in memory. Factoring out layout aspects into composition facilitates reuse. A single set of declarations can be used to form a data structure which is efficiently packed in memory and suitable for single-threaded use as well as a more sparsely laid-out and scalable data structure. The difference is counted in a few characters ($\otimes$ exchanged for $\oplus$ or vice versa).

An alternative to avoid false sharing is to implement splits as actual splits, rather than just pointers into sub-structures. Splitting a pointer of type $c_1 \otimes c_2$ into two pointers typed c_1 and c_2 would keep the original pointer (upcast to a c_1), create a copy of the original object's parts relevant for c_2, and the second pointer would point to the (partial) copy. Section 2.5 below introduces machinery for merging a split pointer back into the original capability, at which point the relevant bits of c_2 would be copied into the original object, and the partial copy discarded. This solution may be favoured if memory is strained, and parallel



List<T> = (add<T> ⊕ del<T>) ⊕ ith*<T>
Cell<T> = Pair<T> ⊗ Pair<T>
List<Pair> = ith*<Pair> = ith*<Cell> and ith*<Cell>

Figure 3. Turning a list of Pairs into two non-interfering iterators to cells. Not to clutter the figure, we omit the fact that a cell could further be decomposed in Get and Set capabilities as was suggested in Section 2.2. (Note: this picture uses colours.)

operations are sufficiently long-running for the copying overhead to be negligible, but clearly works less well with splitting large data structures, *e.g.,* a list. A further down-side to actual splitting is that read–write channels between splits would take an additional indirection hit.

2.5 Splitting, Merging and Jails

The unrestricted use of all available capabilities is always sound from a parallel standpoint, which voids the need to track effects to exclude unwanted non-local behaviour. Instead, a splitting machinery must be introduced to allow capabilities to be lost temporarily, and be regained later. "Lossy splitting", for example turning a pointer to a $\texttt{List}\langle\texttt{Pair}\rangle$ into multiple pointers to $\texttt{ith}^*\langle\texttt{Cell}\rangle$, is a lot less useful unless the original capability can be restored. Below is an example of splitting and merging a pair into two cells:

```
Pair p;
Cell c1, c2 = (Cell * Cell) p; // split pair into two cells
// p is invalidated (e.g., nullified)
... // operate on the cells, possibly in parallel
try {
  p = (Pair) c1, c2; // merge c1 and c2 into a pair
  // c1 and c2 are invalidated (e.g. nullified)
} catch (UnrelatedMergeException ume) {
  // c1 != c2
}
```

The splitting and merging above retains all capabilities to the pair object. However, consider the following code which performs a "lossy split":

```
Cell c;
Get g = (Get) c; // lossy
c = (Cell) g; // will fail because there is no provided Set component
```

Here, the Set capability is forever lost. We turn lossy splitting into non-lossy splitting by the introduction of *capability jails.* A capability jail $j\langle c\rangle$ is a witness to the existence of the capability c, but does not permit any operations on c.

The only use for a capability jail is to restore a capability in a merge, which causes the jailed capability to be consumed. Using jails allows the disjunctive capability $c_1 \oplus c_2$ to be turned into a conjunctive capability $j\langle c_1 \rangle \otimes c_2$ which can subsequently be split in a non-lossy way. In implementation terms, this boils down to the creation of two aliases, one whose static type in the surface language is $j\langle c_1 \rangle$ and one whose type is c_2. Reconstructing a single pointer with the type $c_1 \oplus c_2$ from two pointers typed $j\langle c_1 \rangle$ and c_2 is trivially sound when the pointers are aliases, meaning that regaining capabilities is as cheap as an identity check and simple to do at run-time. When splitting and merging happens locally, the identity check will even be trivial to remove in many circumstances using flow-sensitive analysis. The lossy splitting of the cell above, can be rewritten with jails like so:

```
Cell c;
Jailed<Set> s, Get g = (Jailed<Set> * Get) c; // not-lossy
s.set(); // compile-time rejected
c = (Cell) s , g; // OK
```

Jails and merges work well with exclusive capabilities, but non-exclusive capabilities complicate matters as they are unbounded in number. For example, consider $c_1 \oplus c_2^*$. If we allow $j\langle c_1 \rangle \otimes c_2^*$, then an unbounded number of c_2 pointers may be created. Clearly, a merge back to $c_1 \oplus c_2^*$ requires that all c_2 pointers are invalidated. (Otherwise, narrowing the capability to c_1 is no longer sound since c_2 aliases may exist which can modify shared resources in parallel.) We solve this by wrapping the non-exclusive capability inside an *alias jail*, which must be treated linearly. Alias jails do not restrict operations on its wrapped capability but disallows creating additional aliases to the same capability. Let $e\langle c_2^* \rangle$ denote such an alias jail which must be treated linearly. Now, we can turn $c_1 \oplus c_2^*$ into $j\langle c_1 \rangle \otimes e\langle c_2^* \rangle$, which is splittable and mergable as normal exclusive capabilities.

Note that alias jails do not require uniqueness, just tracking of all aliases isomorphic to fractions in fractional permissions [1]. Thus, we can turn c_2^* into c_2^n for some known n which allows the creation of n aliases.

A less convoluted and arguably more high-level approach is to introduce a scope for splitting to bound the life-time of aliases created from the original value, and only allow merging as the scope exits. This is a simple and straightforward solution, but imposes a structure on the programming that may be too restrictive in practise if splits and merges may occur in different places in a program. For now, we are content with the more general low-level approach as the high-level one is easily added at a later stage.

2.6 Putting it Together 2: A Simple Stack

In Capable, an exclusive capability can only be used by one thread at any given time. Imagine, for example, a single-threaded stack implementation in Capable. It could look something like Figure 4, where we use a `consume` operator for destructively reading a variable to maintain linearity.

A Stack class is composed from Push and Pop capabilities with straightforward semantics. Pushing a new value on the stack requires that the old top

```
class Stack<T> = Pop<T> + Push<T> {
  Link<T> top;
}

exclusive trait Push<T> {
  require Link<T> top;
  void push(T o) {
    top = new Link(consume o,
                   consume top);
  }
}
```

```
class Link<T> {
  Link<T> next;
  T value;
}

exclusive trait Pop<T> {
  require Link<T> top;
  T pop() {
    T temp = consume top.value;
    top = consume top.next;
    return consume temp;
  }
}
```

Figure 4. A simple stack in Capable defined for single-thread use.

is consumed into the new top link. Popping, isomorphically requires that the exclusive capability of the top link's next field is moved into top, unlinking the old top in the process.

The link class introduces an exclusive capability that cannot be split further.

2.7 Aliasing Exclusive Capabilities

So far we have only discussed allowing aliasing when the aliases are safe to use in parallel. Forbidding aliasing is too restrictive for code that uses aliasing to efficiently navigate to places in a data structure. For example, consider a singly-linked list to which we want to be able to prepend and append values. If links are exclusively pointed to, prepending is simple to implement in $O(1)$, but append is $O(n)$ as we are forced to traverse the list at each step to find the last link. Maintaining a last pointer in addition to a first pointer is a preferred solution as it allows both prepending and appending in constant time. However, forcing the use of locks or transactions, or even lock-free mechanisms, for something like a last-pointer, can easily generate unwanted overhead or complicate programming.

We are exploring two possible solutions to this problem, each with its own merit: *alias groups* and *encapsulated effects*.

Exclusive capabilities void the need to track effects—one of their immediate advantages. However, it is possible to break free of this mould and allow multiple aliases to a single object if operations on this object is tracked appropriately using an effect system. This allows the last link, but would rule out parallel operations on `last` and `first` (or `last` and `last` etc.). Furthermore, if the list itself is an exclusive capability, this effect information can be suppressed soundly, limiting the amount of effect propagation in the system. It is thus possible to trade simplicity for aliasing for a limited scope only.

A similar result can be obtained by the notion of an alias group, which is a set of aliases into a common data structure. Operations must chose one variable from the group, and ignore the others. An alias group is similar to an ownership context without nesting. Adding an object to a group requires that all aliases to the object outside the group is destroyed, and moving an object out of a group requires that all its aliases inside the group are destroyed. As long as all group variables are defined in a single lexical scope (for example, `first` and `last` in the case of the linked list example), maintaining these rules is relatively simple.

3 Lock-Free Programming in Gifted

In this section, we discuss the semantics of the "lock-free" subcategory of the safe capability, which was show in Figure 1.

3.1 Checking Non-Exclusive Safe Capabilities

While orthogonal to the core design of the Gifted system, how safe capabilities are checked to be safe in practice, is an interesting question. Simple ways to make sure that a capability provides a basic level of thread-safety is to make all its methods "synchronized" in the Java sense or wrap every method call on it in a transaction. Both these approaches would however likely suffer from scalability issues, deadlocks or livelocks. Other simple examples include immutable data or commutative operations, like insertion of elements into a set.

In this section, we outline a strategy for turning exclusive capabilities into safe capabilities by attempting to support lock-free programming idioms and constructs. These idioms and constructs are captured as statically enforced rules for capability manipulation, with some necessary, but minimal, dynamic involvement.

The main idea for supporting lock-free programming in Gifted surrounds ownership of objects, speculation, and atomic committing of changes. The property enforced by the system loosely described in this section is that exclusive capabilities will stay exclusive, even though there are multiple pointers to them.

A capability from a lock-free trait may be shared across multiple threads. Thus, for a capability inside the trait to be exclusive, it must be guarded against parallel access, or exclusivity can be violated. A naive implementation of such a guard is (atomic) destructive reads to ensure that capabilities stay exclusive to one single thread. This, however, is effectively the same as using locks. (Not to mention unfortunate side-effects as linked data structures fall apart due to insertion of null-pointers.) Instead, we introduce a strong notion of speculation.

3.2 Speculation

In Gifted, the concept of speculation is reified. Fields of exclusive capabilities may be marked as "speculatable" and subsequently allow multiple threads to read them non-destructively. The current **`this`** inside a lock-free trait is speculatable by default. The result of a *speculative read* is called a *stymied pointer*. Stymied pointers represent pointers whose ownership is in flux, and may be transferred between any thread at any point in time. Stymied pointers may only be stored on the current thread's stack. In order to change a field of a stymied pointer, the current thread must *assert ownership* of it. This is done by assigning an exclusive capability to the field using a *compare and transfer* operation (**CAT**), which in Java-like pseudocode can be seen as `CAS(f, oldF, newF) && newF = null`[4]. Logically, it atomically compares `f` to `oldF` and, if the values are equal, updates `f` with the value of `newF`, nullifying `newF` in the process. It is important to note that the nullification of `newF` is performed only if the **CAS** succeeds, and not atomically with the swap. Delayed nullification is sounds as `newF` must be exclusive to the current thread so no one can observe the update between the swap and the consume[5].

Accessing a field of type c when c is some exclusive capability is only allowed on an exclusive target, unless the field is marked to explicitly allow speculation. In an implementation of a parallel stack, only the `top` field needs to be speculated upon. In an implementation of a parallel linked list, all `next`-fields of all links will form a directed path which can be speculated on. For simplicity, we assume that an object has at most one speculatable field.

3.3 Reclaiming Ownership

To store to a speculatable field, the source value must be an exclusive capability, which is consumed. Field reads via stymied pointers yield stymied pointers, so by default, there will be only one exclusive capability to an object, stored *on the heap* in the speculatable field. Stymied aliases are to some respect witnesses to the existence of this original capability in the data structure at some point.

The only way for a thread to get exclusive access of a value in a speculatable field is by writing to the field holding the original exclusive alias. This will necessarily unlink the object from the data structure, and no subsequent attempt by parallel threads **CAT**'ing on the same field will ever succeed, unless the exclusive capability is stored back into the field. Figure 5, Line 7 exemplifies this. This means that we can allow common patterns like `CAT(x,y,z.f)` although the nullification of `z.f` is not performed atomically with the underlying **CAS** when `x=z`, since if the **CAS**-part succeeds, the exclusive capability `x` will have been transferred to the current stack, meaning that `z.f` is stable wrt. concurrent modifications. This avoids inserting null-pointers into "live parts" of a data structure.

3.4 Example: Parallel Stack

For concreteness, look at the implementation of a `Pop` trait for a parallel stack implementation in Figure 5. The `top` field is the subject of possible speculation by several parallel threads. The variable `oldTop` holds a stymied alias of the `top`-most

[4] A compare and swap (**CAS**) operation *atomically* compares the contents of a memory location to a given value and, if they are the same, updates the contents of that memory location to a given new value. A typical use of CAS involves reading a variable x to obtain a value v, then speculatively performing an operation on v to yield a new value v'. Now, if x still holds v, the speculation is still valid, and we can go ahead an "publish the results" of the speculation—v'.

[5] For brevity, we omit a discussion on dealing with ABA problems, which we do using pointer tags to track values which are "in flux".

```
 1  lockfree trait Pop {
 2    require speculatable Link top;
 3    Object pop() {
 4      Link oldTop;
 5      do {
 6        oldTop = speculate top; // oldTop is stymied
 7      } while (CAT(top, oldTop, oldTop.next) == false);
 8      // From here on, oldTop is not stymied anymore!
 9      return consume oldTop.value;
10    }
11  }
```

Figure 5. Lock-free implementations of the `Pop` trait from Figure 4.

link at the time of the speculation. We are not allowed to read `oldTop.value` in lines 4–7 because value is not a designated target for speculation. Notably, the **`this`** variable is stymied, which forces us to assert ownership when updating `top`—the **`CAT`** on line 7. If `top == oldTop` succeeds, then the current thread asserts ownership of the current **`this`** and further transfers the exclusive capability in the `top` field to itself, since the shared `top` field will be overwritten.

Thus, `oldTop` is exclusive in the last compartment of the **`CAT`**, which allows the (destructive) read of the `next` field to transfer its contents into `top`. Note that since `next` is not speculatable, it contains an exclusive capability, which is required to store values in `top`. Also note that since `oldTop.next` was destructively read, the field can no longer be used to reach the internals of the stack—`oldTop` has been unlinked from the data structure.

A simple flow-sensitive analysis will be able to identify that `oldTop` is exclusive from line 8 forward. The treatment of `oldTop.value` is therefore isomorphic to `oldTop.next`.

3.5 Support for Structures with Multiple Contention Points

State-of-the-art lock-free implementations of single-linked structures with **`CAS`** use low-level trickery [9] to be able to separate logical deletion of a link from a chain from its actual unlinking without the need for a **`CAS`** that works on multiple memory locations. This involves using insignificant bits of pointer addresses to keep all relevant information in a single word of memory.

Logical deletion involves marking an object to be deleted, and operations like insert to check that a marked object is not used to insert a new link (*e.g.,* no `m.next = newLink` when `m` is a marked object) since the actual unlinking will remove the object from the chain, causing `newLink` to be lost. (See [9] for nice pictorial examples.)

Gifted supports marking objects for logical deletion with a few simple rules. Objects with a "speculatable field" automatically get a `mark()` operation which signals the logical deletion of its exclusive capability from the (necessarily singular) data structure of which it is a part. Marking an object in Gifted causes subsequent

```
 1 lockfree trait Push {
 2   require speculatable Link top;
 3   void push(Object o) {
 4     Link newTop = new Link(consume o, null);
 5     do {
 6       Link oldTop = speculate top; // oldTop is stymied
 7       newTop.next = oldTop; // newTop *becomes* stymied!
 8     } while (CAT(top, oldTop, newTop) == false);
 9   }
10 }
```

Figure 6. Lock-free implementations of the `Push` trait from Figure 4.

`CAT`s on the field holding the "marked bit" to fail[6]. This means that a marked object is effectively immutable with respect to speculation. This is important because this prevents parallel threads from extending the data structure by building on an object which is slated for removal.

In addition to the `mark()` operation, objects with speculatable fields automatically get a `nextUnmarked()` operation, which follows a directed path of marked fields and returns the first unmarked object. The key insight is that since `next`-fields on marked objects are effectively immutable, the result of this operation is stable. Therefore, unlinking in a linked list such as `CAT(p, n, n.nextUnmarked())` is entirely stable: the result of `n.nextUnmarked()` will never become invalidated. Since this will invariably unlink `n` from the data structure, it's ownership is transferred to whichever thread succeeds in carrying out this `CAT`.

3.6 Two Additional Pieces of Magic

Operations such as `CAT(x, y, y.f)` introduce null-pointers in data structures which is visible to all threads speculating on `y`. This is sound, yet not ideal, for example if null-pointers are used to denote *e.g.,* end of list. Allowing `CAS(x, y, y.f)` without nullifying `y.f` avoids this problem, but instead introduces a problem duplication of exclusive capabilities. This problem can be solved by changing the type of `y` following the `CAS` and remove any capability that mentions `f`. This effectively removes `f` from the object when accessed through `y` and allows freely using `y` and even re-inserting `y` in the data structure.

Figure 6 shows a lock-free implementation of `push` in Gifted. On line 6, we speculatively read `top`. For speculation to be sound, only one thread can be allowed to *publish* the results of the speculation, in this case storing `top` back in the structure again, but linked to from a fresh link. We again rely on the flow-sensitive analysis: we only allow $x.f = v$ when v is stymied if we can determine that v will only be published after v is transferred to the current thread. In other words, $x.f = v$ has the side-effect of making x dependently stymied on v. Since stymied pointers exist on the stack only, this prevents publishing a speculative

[6] Due to the mark-bit present in the field, but not in the stymied alias.

results (an "unconfirmed speculation"). To lift the stymieing of x, a successful assertion of ownership of v is required.

In our example, `newTop` becomes dependently stymied on `oldTop` on line 7. However, success of `top == oldTop` implicitly performed on line 8 allows `newTop` to regain its exclusive type, as required to be able to store the value in `top`.

4 A Brief Account of Related Work

This section does not try to give a complete summary of the research in the fields of alias prevention and control, or capabilities. Instead, we compare this paper to recent similar work to explain its context.

Our work is closest in spirit to work by Caires and Seco [3] and Militão et al [11]), as well as work by Pottier et al [13]. Perhaps interestingly, we arrived at similar designs from a somewhat different angle, namely our past work on read-onlyness and immutability [12], ownership types and ownership transfer [5]. Capable owes some of its design to our on-going work on Gift [2], where objects are unaliased from the start and can become gradually more aliased. Gift does not use capabilities and relies on an effect system to guarantee deterministic parallelism, which is something that Capable tries hard to avoid. We expect to reuse parts of the Gift system for the work on aliasing of exclusive capabilities in Section 2.7.

Capable is partially reminiscent of Pottier and Protzenko's Mezzo [13]. While Capable aims to simplify parallel programming, Mezzo focuses on detailed reasoning in a sequential setting using predominantly linear permissions, similar to capabilities. Mezzo's powerful machinery for linear permissions could in principle be added to Capable's exclusive capabilities. In Mezzo, types and permissions are unified, and linear treatment of permissions enable type-state like static protocol checking and support for powerful patterns including type changes, late initialisation, etc. Some support for sharing mutable objects exists through mostly dynamically checked means (notably techniques very similar in spirit to Section 2.7). Parallel programming is supported through locks.

Caires and Seco [3] (later continued by Militão et al [11]) use capabilities for reasoning about interference caused by aliasing or concurrent accesses to aliased data. Their type system is more fine grained and the type of a capability can also express a temporal protocol of its use, for example that a data type must be initialized before use. They handle aliasing capabilities by either checking that there is no possible interference or by using locks to synchronize accesses to shared data. We believe that there are many similarities between this work and our's and that there is merit to both levels of granularity. What we lack in expressiveness, we make up by having a less complicated system with less syntactic and typechecking overhead.

Haller and Odersky [8] use capabilities to enforce uniqueness and borrowing constraints in an actor based framework. Since no aliasing capabilities are ever used in parallel, they are all exclusive (and therefore trivially safe). We extend on this work by also allowing safe, aliasing capabilities.

Turon's Reagents [14] simplify lock-free parallel programming by providing a set of powerful abstractions for speculation and publication. Reagents simplify parallel programming and thereby improves program quality, without any need for extended static checking. An important part of our road-map is to marry Reagents with Capable capabilities. For Capable, this will provide a superior mode for implementing lock-free algorithms in a composable fashion. For Reagents, we will be able to exclude problems that can arise due to unintentional aliasing.

5 Conclusion

Capabilities provide powerful means to control aliasing while avoiding some of the common downsides to existing work on alias management: imposing a structure on the program which does not work well for all programs, or reporting effect information which propagate through the code and interplays badly with polymorphism, reuse etc. The Capable capability model seems flexible enough to allow mixing and matching several means of parallel programming, especially with respect to control mechanisms and static checking, and also capture relevant memory behaviours without destroying reuse.

References

1. J. Boyland. Checking Interference with Fractional Permissions. In *SAS* 2003.
2. S. Brandauer, E. Castegren, and T. Wrigstad. Gift: May-alias types for gradual sharing. In preparation.
3. L. Caires and J. Seco. The type discipline of behavioral separation. In *POPL* 2013.
4. N. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple Ownership. In *OOPSLA* 2007.
5. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.
6. D. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In D. Clarke et al., editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *LNCS*, pages 15–58. Springer, 2013.
7. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, March 2006.
8. P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.
9. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In DISC '01.
10. J. Hogg, D. Lea, A. Wills, D. de Champeaux, and R. C. Holt. The Geneva convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
11. F. Militao, J. Aldrich, and L. Caires. Rely-guarantee protocols. 2014. To appear at ECOOP'14.
12. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, Uniqueness, and Immutability. In *TOOLS*, 2008.
13. F. Pottier and J. Protzenko. Programming with permissions in Mezzo. In *ICFP*, 2013.
14. A. Turon. Reagents: Expressing and composing fine-grained concurrency. In *PLDI*, 2012.