Kappa: Insights, Current Status and Future Work[1]

Extended abstract presented at IWACO'16

Elias Castegren
Uppsala University
elias.castegren@it.uu.se

Tobias Wrigstad
Uppsala University
tobias.wrigstad@it.uu.se

Abstract

KAPPA is a type system for safe concurrent object-oriented programming using reference capabilities. It uses a combination of static and dynamic techniques to guarantee data-race freedom, and, for a certain subset of the system, non-interference (and thereby deterministic parallelism). It combines many features from previous work on alias management, such as substructural types, regions, ownership types, and fractional permissions, and brings them together using a unified set of primitives.

In this extended abstract we show how KAPPA's capabilities express variations of the aforementioned concepts, discuss the main insights from working with KAPPA, present the current status of the implementation of KAPPA in the context of the actor language Encore, and discuss ongoing and future work.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Type systems, Language Implementation, Capabilities, Traits, Concurrency, Object-Oriented

1. Introduction

The last few decades have seen considerable interest in type systems for controlling aliasing and interference, ranging from approaches that constrain the structure of a program, like different flavours of ownership types and universe types, via linear types and type qualifiers, to more descriptive techniques like effect systems.

While reasoning about aliasing in a sequential setting is important for functional correctness, the increasing ubiquity of multi-core and many-core architectures makes controlling aliasing even more important. An unfortunate scheduling of two threads sharing mutable state could cause data-races, which leads to problems like lost updates, corrupted data and unwanted non-determinism.

This paper overviews the KAPPA type system for concurrent and parallel programming, how it leverages a capability-based way of thinking, and how it integrates with object-oriented programming. We discuss the current stable KAPPA system, the current and ongoing work on extending KAPPA, and finally dare to dream about the future.

2. KAPPA and the Past

In this section we introduce KAPPA and explain how it expresses concepts from a wide variety of previous work on alias management. A more thorough treatise of the system can be found in a current paper to be presented at ECOOP'16 [12]. Some initial sketches from a previous IWACO paper are also available [10].

The starting point for KAPPA is the unification of references and capabilities. A capability is a token that grants access to a particular resource. In KAPPA these resources are objects, parts of objects, or entire object aggregates (an object containing other objects). Capabilities present an alternative approach to tracking and propagating computational effects to check interference: capabilities assume exclusive access to their governed resources, only permit reading, or follow some protocol that allows resolving potential conflicts. Thus, holding a capability implies the right to use it fully without fear of uncontrolled data-races. This importantly shifts reasoning from use-site of a reference to its creation-site. Granting and revoking capabilities corresponds to creating and destroying references.

2.1 KAPPA in a Nutshell

In KAPPA, capabilities are introduced via traits. A trait can be thought of as an abstract class whose *fields* are abstract and must be provided by a concrete subclass. Another way to think about traits is as Java-style interfaces that can name fields and provide implementations of methods. The following code defines two traits for reading and incrementing an integer field:

```
trait Inc
  require var cnt : int
  def inc() : void
    this.cnt = this.cnt + 1

trait Get
  require val cnt : int
  def get() : int
    return this.cnt
```

Both traits *require* a field `cnt`, meaning that if they are included by a class that provides such a field, the traits will provide their respective methods. A class that does not provide all the required fields of its traits does not typecheck. Note the difference between **var** fields which may be updated, and **val** which may not.

A KAPPA capability is a trait annotated with a *mode*, that controls how the capability gains exclusive access to the underlying object. For example, **linear** `Inc` is a capability that gives access to the `inc` method of its governed resource. The linear keyword means that it must be treated linearly (*i.e.,* never have more than one alias). It can therefore only ever be accessed by one thread at a time. KAPPA provides a number of modes:

linear – the capability must be treated linearly;

thread – the capability can be aliased freely, but aliases are restricted to a single thread;

[1] This work was partially funded by the Swedish Research Council project Structured Aliasing, the EU project FP7-612985 Upscale (http://www.upscale-project.eu), and the Uppsala Programming Multicore Architectures Research Centre (UPMARC).

locked – interactions with the capability will be wrapped in acquiring and releasing of a lock;

read – the capability only provides reading operations;

subordinate – the capability is strongly encapsulated inside some object and inherits protection from data-races from it; and

unsafe – the capability provides no protection of its own, *i.e.,* a normal reference in most object-oriented languages.

The **linear** and **thread** modes are *exclusive* modes as such capabilities are always exclusive to a single thread. (Although **linear** capabilities might be transferred between threads, at any point in time at most one thread can access it.) The **locked** and **read** modes are *safe* modes as they are always safe to share between threads, either because accesses will be serialised by using locks, or because all provided operations will only perform reads on the underlying object.

The **subordinate** mode is special in the sense that it doesn't provide any protection mechanisms of its own, but relies on being encapsulated by some other capability. We call capabilities that can provide protection for other capabilities *dominating* capabilities.

Unsafe capabilities can either be thought of as objects that have no automatic protection of their own and must therefore *e.g.,* be manually locked before usage, or as an escape hatch from the type system when some aliasing pattern known to be safe cannot be expressed or when data-races can be allowed. So far we have chosen the former solution, requiring a Java-style **sync** block (in which the **unsafe** capability can be viewed as a **locked** capability).

As usual in a trait-based system, KAPPA constructs classes and types by composing traits, or more precisely capabilities. There are two forms of composition: disjunction ($\oplus$) and conjunction ($\otimes$). If `A` and `B` are capabilities, their disjunction `A` $\oplus$ `B` provides the disjoint union of the methods of `A` and `B` and requires the union of their field requirements. Intuitively, the disjunction `A` $\oplus$ `B` can be used as an `A` *or* a `B`, but not both at the same time (*i.e.,* not in parallel).

The conjunction `A` $\otimes$ `B` also has the same requirements and provides the same methods as its constituents, but is only well-formed if `A` and `B` do not share mutable state which is not protected by concurrency control (in other words, a shared field must be a **val** field containing a safe capability). This means that `A` $\otimes$ `B` allows `A` and `B` to be used in parallel.

The following snippet declares a linear counter class using the traits defined earlier:

```
class LinearCounter = linear Inc ⊕ read Get
  var cnt : int
```

The class `LinearCounter` provides the field `cnt` required by the included traits. Note that the composition `Inc` $\otimes$ `Get` is not allowed as the two traits share the mutable field `cnt` – concurrent calls to `inc()` and `get()` would race on `cnt`.

Since `LinearCounter` is composed from a **linear** capability, any variable of this type must also be treated linearly. Through upcasting, the mutating `Inc` capability can be forgotten, leaving only **read** `Get` which allows sharing the underlying object across threads without dynamic concurrency control (the remaining **read** capability only performs reads).

The same traits can be used to declare a counter class that uses locks for protection instead:

```
class SynchedCounter = locked Inc ⊕ read Get
  var cnt : int
```

Since all sub-capabilities of `SynchedCounter` are safe, the full type is also safe, and a variable of this type is safe to share across threads. However, since the `Get` capability shares state with the `Inc` capability, calls to `get` must also be synchronized via locking. This can be implemented using a readers-writer lock, with a static guarantee that readers will not write (as a **read** capability may only use **val** fields).

To exemplify conjunctive capabilities, the following snippet implements a pair of counters:

```
trait Fst
  require var fst : LinearCounter
  def getFst() : int
    return this.fst
  def incFst() : void
    this.fst.inc()

trait Snd
  require var snd : int
  ... // symmetric to Fst

class LinearPair = linear Fst ⊗ linear Snd
  var fst : int
  var snd : int
```

As `Fst` and `Snd` do not share any mutable state, their conjunction is well-formed, and a capability of type `LinearPair` can be *unpacked* into its constituents:

```
var p = new Pair(2,3);
let (fst : Fst, snd : Snd) = consume p; // 1
finish{
  async{fst.incFst()}
  async{snd.incSnd()}
}
p = consume fst ⊗ consume snd; // 2
```

At (1) the `Pair` is unpacked into two capabilities `fst` and `snd`. The **consume** operation denotes a destructive read which nullifies its target—this is required (in general) to maintain uniqueness of **linear** capabilities. Note that `fst` and `snd` are aliases of the same object, but that operating on them in parallel is safe. At (2) the original capability is restored.

This section gave a brief overview of some of the features of KAPPA. A more thorough presentation, including more examples, can be found in our ECOOP'16 paper [12]. The following sections will expand on the presented features and show how they can be understood by comparing to related work.

2.2 Subordination and Ownership

Subordinate capabilities denote resources which are safe to access because all references to them are hidden inside some other capability. The subordinate capability thus inherits the protection of the other capability. Capabilities which are able to offer protection for subordinate capabilities are called dominating capabilities. Some capabilities are neither subordinate nor dominating – for example, a **read** capability may not contain any subordinate state.

The subordinate mode is similar to **rep** and **owner** in ownership types, but KAPPA does not emphasise nesting strongly. In its most simple form, the heap is shallowly partitioned into a set of dominating or **read** capabilities[1] and their subordinate state. It is however possible to form combinations of subordinate and dominating capabilities, and thereby create deeper levels of nesting. For example, a capability **linear** $\oplus$ **subordinate** is unaliased, and may additionally not be transferred outside of its enclosing aggregate.

In KAPPA, a reference from object o_1 to object o_2 requires that (1) o_2 is some capability which knows how to handle concurrent accesses, or (2) o_2 is subordinate state inside the same enclosure as o_1 (including the case where o_1 "owns" o_2). Note that this precludes outgoing references, *i.e.,* if o_1 is subordinate to o_3 and o_3 and o_2 are siblings. This is required to preserve data-race freedom.

[1] Using ownership types parlance, these capabilities are in **world**

Consider a situation as depicted in Figure 1. Notably, threads (here, T_1 and T_2) may operate in C and D concurrently, *e.g.,* some thread enters D and then forks off a task that contains C. Allowing outgoing references would allow the thread in C to follow the reference in A to B, and the thread in D to access B—breaking data-race freedom. Note: if B is a dominating capability, the reference is legal under (1) above.

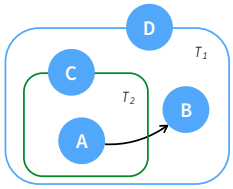

Figure 1. Outgoing reference.

2.3 Subordination and Trait-based Reuse

An important property of KAPPA is that the implementor of a trait can assume that the executing thread has exclusive access to the required resources. This property allows the same trait to be given different modes for different usages, which improves trait-based reuse for different concurrency scenarios.

With the exception of the **read** mode, the key restriction that allows a trait to be given any mode is that methods do not assign **this** outside of the enclosing aggregate. This is achieved by typechecking traits without manifest modes with **this** as a **subordinate** capability. This means that **this** may only be passed to methods of capabilities that are also **subordinate**, and only returned from a method if the caller is **subordinate**.

It is reasonable to ask whether or not this default is too restrictive. Vitek and Bokowski [22] show that 84–95% of methods in `java.util` and `java.awt` (JDK 1.1) are anonymous methods, meaning that they only use **this** to access instance variables directly. Wrigstad's *et al.* Loci [23] uses similar restrictions and successfully applies its tool to a 50 KLOC Java program without encountering problems due to leaking **this**. This suggests that defaulting to **subordinate** is reasonable. If some application requires **this** to be leaked outside of its enclosure, adding a manifest mode ensures the concurrency control necessary to avoid potential data-races (at the cost of disallowing the trait to be given any mode at use-site).

2.4 Linearity and Borrowing

The **linear** capabilities are basically the same as the linear or unique references found in many other languages [1, 4, 5, 7, 9, 13, 17, 20] (but see [11] for how this may diverge). Composing two linear capabilities `A` and `B` in a conjunction `A ⊗ B` however allows us to create two linear aliases that access disjoint parts of an object.

A trait whose linearity is declared in conjunction with its composition, which has been the case in all examples so far, is free to alias **this** internally (*cf.,* § 2.3), meaning they are *externally unique* [13]. The externally unique aggregate contains the subordinate fields of the **linear** capability. A trait which is *manifestly* declared as linear, knows of its uniqueness and can therefore leverage this knowledge[2], at the price of disallowing internal back-pointers.

Borrowing linear values follow standard rules: the value at the end of an *all linear* path `x.f.g...h` can be temporarily placed on the stack and reinstated once the stack-frame is popped, as long as no prefix of the path is accessed during the borrowing (a prefix of the path could be use to create an unsafe alias of the borrowed value). Borrowing is denoted by an "**S**-box" wrapping a type, *e.g.,* **S**`(Pair)`. In conjunction with borrowing, linearity may be relaxed to **thread**, which allows freely aliasing the value, but not passing it off to any other threads.

The simplest case of borrowing is when the path of linear values is a singleton variable `x`. We call this type of borrowing *forward* borrowing, as we are passing a stack-bound variable to another method or function. Relaxations can be supported in a fine-grained manned, as exemplified below:

```
def foo(x:S(linear Fst ⊗ thread Snd)) : void ...

foo(p);
```

Here, the pair `p` is passed non-destructively (note the lack of a **consume**) to `foo()` and in the process is unpacked into a linear `Fst` part and an aliasable `Snd` part. The original variable `p` will be buried on the stack frame below for the duration of call to `foo()`.

We call borrowing when the path of linear values is longer than one *reverse borrowing*. This allows non-destructive reads of linear fields into stack-bound values. For example, the following `Cell` class uses reverse borrowing to return the `elem` field without destroying it – which lets the `Accessor` trait have the **read** mode.

```
read trait Accessor<linear T>
  require elem : T
  def get() : S(T)
    return this.elem

trait Mutator<linear T>
  require elem:T
  def set(e:linear T) : void
    this.elem = consume e

class Cell<linear T> = linear Mutator<T> ⊗ Accessor<T>
  var elem:T
```

The trick to sound reverse borrowing is to require the target to be linear and to prevent multiple borrowings of the same value, *e.g.,* by disallowing storing result in a variable. Thus, if `c` is a `Cell<Pair>`, we may perform `c.get().setFst(42)` or `foo(c.get())` knowing that no other on-going computation will be able to witness the alias in `c.elem` before these expressions are fully evaluated and uniqueness has been restored.

2.5 Composition and Regions

Regions provide a means of dividing an object up into disjoint parts that can be operated on in parallel. For example, in Deterministic Parallel Java, each field of a class belongs to a region and methods are annotated with effects to show which regions they access [2]. Methods with non-overlapping effects can safely be run in parallel.

In KAPPA there are no explicit regions and no effect annotations. Instead, variable requirements in traits, and trait composition, allows regions to be inferred. Remember that a trait requires fields and that fields can be **var** (mutable) or **val** (immutable). A safe over-approximation of a trait's methods is to assume that they write to all of the trait's **var** fields and read from all of the trait's **val** fields. For example (*cf.,* § 2.1):

– The methods in `Fst` sees `fst` as a **var** field, and can therefore therefore be assumed to write to this field.

– The method in `Get` sees `cnt` as a **val** field, and can therefore be assumed to read this field.

Since two traits on opposing sides of a conjunction `A ⊗ B` may not share any unprotected mutable state, their mutable fields are conceptually in different regions. For example, from the composition `Fst ⊗ Snd` of the class `LinearPair` we can derive two disjoint regions – one for the field `fst` and one for the field `snd`. By deriving effects as above we can see that the methods in one trait only write to the region derived from that trait, and so two methods from different traits have disjoint effects and can be run in parallel. This is exactly the behaviour allowed by capability conjunction.

[2] This strong form of uniqueness enables strong updates, type-safe dynamic re-classification [15], type state [14, 19], etc. This is a direction of future and on-going ([11]) work.

By reasoning at the level of traits rather than individual methods, we are checking for interference at a higher level of abstraction. This gives us the benefits of a full region and effects system, but also lets us avoid the overhead that comes with annotations.

2.6 Unpacking and a Taste of Fractional Permissions

Fractional permissions [6] enable mediating between a single mutable (full) permission and several read-only (fractional) permissions. The intuition is that the sum of the fractions should always add up to a full permission. This allows expressing patterns where a single thread mutates some object and distributes it to some other threads which only read it. When all threads are done reading (and the fractions are reassembled), the original thread can perform a new update.

In KAPPA there are no explicit fractions or permissions, but the single writer–multiple readers pattern can be simulated by forgetting and restoring (parts of) **linear** capabilities. The capability **linear** `Inc` $\oplus$ **read** `Get` of the class `LinearCounter` (*cf.,* § 2.1) conceptually holds the full permission to mutate the counter. By upcasting the capability to **read** `Get` we get a capability that may be arbitrarily shared, but at the same time we lose the mutating capability forever.

To be able to restore the full capability, we can temporarily hide the mutating capability and distribute the safe capability among the reading threads, and then restore the mutating capability after we know that all the safe capabilities have gone out of scope. KAPPA provides a scoped construct for this:

```
let c = new LinearCounter();
...
bound c as g : S(read Get) in {
  finish{ // c is hidden...
    async{foo(g)}
    async{bar(g)}
    async{fr0b(g)}
  }
} // ...until g has gone out of scope
c.inc();
```

With actual fractional permissions it would be possible to reassemble the full permission in a different location than where it was split up. To support this in KAPPA we would need some kind of "enumerated unpacking" to track the number of aliases created. Currently we only packing and unpacking in different locations for linear capabilities.

In addition to the single writer–multiple readers pattern, KAPPA also allows multiple disjoint writers (through unpacking of capability disjunctions) and multiple overlapping writers (through capability disjunctions that share safe capabilities) to the same object.

3. KAPPA and the Present

In this section we present the current status of the implementation of KAPPA in the context of the actor language Encore [8]. We also discuss ongoing work to allow capabilities that use protocols from lockfree programming to safely share data.

3.1 Encore in a Nutshell

Encore [8] is an object-oriented programming language for parallel and concurrent programming. Encore achieves concurrency by using active objects – objects with their own (conceptual) thread of control, communicating asynchronously with each other through message passing. Encore additionally provides means for pipeline-style parallelism aimed at big data-style computations [16]. The KAPPA system is integral to avoiding data-races in Encore.

In Encore, classes are active by default. The following program will create two active `Greeter` objects that will print their id ten times each in non-deterministic order:

```
class Greeter
  id : int
  // A constructor method
  def init(id : int) : void
    this.id = id
  def greet() : void
    print("Hello, my id is {}", this.id)


class Main
  def main() : void {
    let g1 = new Greeter(1);
    let g2 = new Greeter(2);
    for i in [1..10] {
      g1.greet();
      g2.greet();
    }
  }
```

Calling a method on an active object immediately returns a future which will be fulfilled with the return value of the method call when the active object has processed the message. Here is an implementation of an active counter class:

```
class ActiveCounter
  cnt : int
  def inc() : void
    this.cnt = this.cnt + 1
  def get() : int
    this.cnt

class Main
  def main() : void {
    let c = new ActiveCounter();
    c.inc();
    c.inc();
    let v = c.get(); // v : Fut int
    print(get v); // Blocks until fulfillment, then prints 2
  }
```

Passive objects (without its own thread of control) are created from passive classes. These objects behave just like regular objects, with synchronous method calls:

```
passive class Counter
  cnt : int
  def inc() : void
    this.cnt = this.cnt + 1
  def get() : int
    this.cnt

class Main
  def main() : void {
    let c = new Counter();
    c.inc();
    c.inc();
    let v = c.get(); // v : int
    print(v); // Prints 2
  }
```

3.2 KAPPA and Encore

As all active objects are running in parallel, every shared passive object is a potential data-race waiting to happen. We remedy this by integrating KAPPA with the passive classes of Encore. As a first step, we have implemented support for **linear**, **read** and **subordinate** capabilities and their composition. As active objects already provide a way to serialise concurrent accesses to data, we have so far excluded

locked capabilities. Passive objects shared between active objects must therefore be read-only or have non-overlapping capabilities so that different threads access disjoint parts of the object.

As with most actor systems, Encore's active objects are opaque and should encapsulate their representation. With KAPPA we can enforce this by using **subordinate** capabilities for the representation of an active object, meaning these objects cannot be passed outside of the aggregate of the active object. By letting **subordinate** be the default mode, mode annotations on traits will only be necessary for objects that will be shared between active objects.

In ongoing work, we are looking at a closer integration of KAPPA and Encore by replacing active classes by an **active** mode on capabilities. This allows using the same traits to create active data as passive data. For example, the active counter class from § 3.1 could be declared as:

```
class ActiveCounter = active Inc ⊕ active Get
  cnt : int
```

It is interesting to ponder the difference in semantics between types such as **active** ⊕ **active** and **active** ⊗ **active**. The natural interpretation of first type seems to be different traits constructing an actor, while the latter opens up for actors with parallel capabilities, and possibly several message queues. Similarly, the type **locked** ⊗ **active** or **locked** ⊕ **active** could denote an actor with a separate "priority channel", that stops the actor between messages to let some other thread access its state. The type **linear** ⊕ **active** could be used for an actor whose linear capability contains the initialisation methods, which can later be forgotten (using an upcast), gaining the ability to alias and share the reference across threads. Working out similar details for other combinations of capabilities is a direction for future work.

3.3 KAPPA and Optimistic Concurrency Control

An alternative to wrapping accesses in locks is to use some form of optimistic concurrency control. Software transactional memory [21] should be easy to integrate with KAPPA on the surface – an **atomic** capability wraps accesses in transactions, and rolls back on conflicts. Another form of optimistic concurrency is found in lock-free programming, which uses compare-and-swap (CAS) or similar atomic primitives to avoid blocking and handle data races.

In ongoing work we have explored a type system design for lock-free programming in KAPPA based on CAS, which requires a principled relaxation of linear capabilities to allow several threads to operate on a "linear" value concurrently. Our key change to make this possible is the separation of ownership and reference – an object can be arbitrarily aliased as long as at most one of the references can access the linear resources of the object. This way ownership remains linear but can be transferred between aliases.

By confining these relaxed linear references to **lockfree** capabilities, this extension can be used together with the rest of KAPPA without requiring any changes to the existing type system. The extended KAPPA type system guarantees data-race freedom, even in the presence of lock-free data structures such as stacks, queues and lists [11].

4. KAPPA and the Future

Among the most important future work are case studies verifying the practical usefulness of KAPPA. This is something we indend to do as soon as our implementation reaches a sufficiently stable state. In this section we look at where the development on KAPPA is going, and discuss some of our ongoing projects.

4.1 The Grand Scheme of Things

The different kinds of capabilities can be placed in a hierarchy as in Figure 2. The three top-level categories are the *exclusive* capabilities,

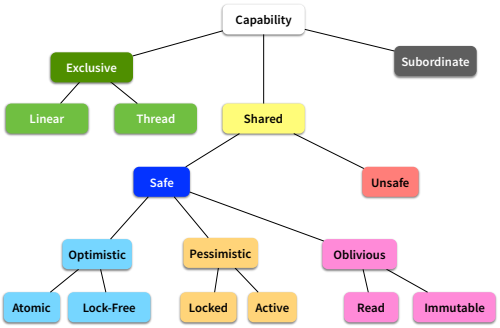

Figure 2. Capability hierarchy.

i.e., the **linear** and **thread** capabilities which are always exclusive to a single thread; the *shared* capabilities, *i.e.,* all the capabilities that may be aliased across threads; and the **subordinate** capabilities which rely on getting protection from some other capability.

The shared capabilities are categorised into the *safe* and **unsafe** capabilities, and the safe capabilities may further be grouped depending on the kind of concurrency control they provide. The *optimistic* capabilities use techniques where threads access the same data without synchronisation following some protocol that allows conflicts to be resolved, *e.g.,* using transactional memory or lock-free programming patterns (*cf.,* § 3.3). The *pessimistic* capabilities serialise all accesses, *e.g.,* by using locks or by wrapping state in an actor whose message queue decides the order of operations. The *oblivious* capabilities do not need any dynamic concurrency control because they only provide non-racy operations. These are the **read** and **immutable** ("deeply **read**") capabilities.

A lot of the future work on KAPPA concerns extending this hierarchy with new kinds of capabilities and reasoning about their interactions and compositions. Some of the compositions mentioned in this presentation are straightforward, but it is for example not obvious what it would mean to compose a **locked** capability with an **atomic** capability that uses transactional memory. The hierarchy also suggests a kind of bounded polymorphism between the different modes. Abstracting over the safe modes (using a **safe** annotation) gives a sort of *polymorphic concurrency control* – code that safely uses some data, agnostic to the kind of concurrency control provided by the underlying capability.

An alternative view of the **unsafe** category is one where parts of a program is proven data-race free by some other means (they could be thought of as a **verified** capability). This way, Kappa could interface with (possibly external) modules verified by some other technique, as long as this technique does not require changes to the rest of the KAPPA program. Currently, we allow using **unsafe** capabilities as long as all accesses are synchronized via locking [12], but any other means of achieving data-race freedom would work.

4.2 Arrays

Operations on arrays is a natural thing to want to parallelise, and we are currently extending KAPPA to handle arrays. Just as with the other capabilities we are interested in expressing aliasing patterns of arrays that are safe from data-races, either because different threads are known to access disjoint parts of the array, or because the overlapping operations are safe to perform in parallel.

Analogous to how a `A ⊗ B` may be split into its constituents since `A` and `B` do not share mutable fields, an array can be thought of as a conjunction of capabilities to access each index of the array. The simplest *horizontal* split takes an array $[e_0, \ldots, e_{n-1}]$ and splits it into two arrays $[e_0, \ldots, e_{k-1}]$, $[e_k, \ldots, e_{n-1}]$ which may be operated on in parallel. Implementation-wise, these two arrays

are aliases where accesses to the latter one are implicitly offset by k. We are also looking at more advanced splitting, such as array slices and stencils.

If the type of the elements of an array is a conjunction $\texttt{A} \otimes \texttt{B}$, an array of type $[\texttt{A} \otimes \texttt{B}]$ may be split *vertically* into two arrays of type $[\texttt{A}]$ and $[\texttt{B}]$. Since the rules of conjunction allows `A` and `B` to be operated on in parallel, the elements of the two aliasing arrays can be accessed concurrently. However, since two threads may be accessing the same indices, the arrays themselves must be turned immutable for the duration of the split. Combinations of horizontal and vertical splitting are possible.

4.3 Value Types

Many functional languages don't need to worry about data-races since they use value semantics for all data, copying values on updates rather than passing them around by reference (which is prone to data-races). In an object-oriented setting, reference semantics often comes more naturally, but there are times when using value semantics makes sense (for example strings in Java and Encore). Other languages, for example C and C++, allow programmers to chose between value semantics and reference semantics for objects and structs. Value semantics also avoid pointer indirection, which allows certain memory optimisations.

KAPPA facilitates trait-based reuse, and we would like to be able to express value types using the same mechanisms. For this we are envisioning another safe mode, the **value** mode, which denotes a capability that is safe to share across threads because any modifications made to it will use value semantics. For example, consider a value semantics version of the counter from § 2.1:

```
class ValueCounter = value Inc ⊕ read Get
  var cnt : int
  ...

let c = new ValueCounter(5);
finish {
  async{c.inc()} // Copy the counter before updating
  async{print(c.get())} // Will always print 5
}
print(c.get()); // Will always print 5
```

Calling `inc` on `c` first copies the counter and then updates this copy. Calling `get` leaves the object unchanged and incurs no extra copying. Calling a mutating method (*e.g.,* a method from a non-**read** capability) on a **value** capability could be understood as the following desugaring:

```
c.inc();
→
let c' = c.clone();
c'.inc();
... // substitute c' for c in this code
```

Interestingly, the depth of the cloning of **value** capabilities depends on the types of the fields in the underlying objects. All **subordinate** and **linear** objects would need to be deeply cloned, as sharing them between copies of an aggregate would be racy (**subordinate** objects assume that they are encapsulated in a *single* object; **linear** objects assume that there is at most one reference to them). Safe capabilities on the other hand need only be copied by reference as accesses to these are free from data-races.

To simplify reasoning about which objects might be cloned under the hood, a reasonable constraint is to require that a trait with the **value** mode only contains fields of safe capabilities, which would not need cloning (except for nested **value** capabilities, but these could be cloned "on demand" depending on which parts of the nested state was mutated). This is similar to how traits with the **read** mode can only contain safe capabilities in **val** fields.

4.4 Ownership Declassification

In ownership types, direct references to internal objects must be banned to protect encapsulation. In KAPPA, encapsulation is a means to reason about the protection offered by a dominating capability. However, in the spirit of ownership declassification [3, 18] we might consider returning a pointer to a subordinate capability outside of its dominating capability by changing its mode from **subordinate** to a safe one.

As an example, imagine a **locked** linked list capability with subordinate links. We may return an alias to a link in the list to outside of the list, for example to some iterator object, if the mode of the alias' type is changed to **locked**. Dynamically, method calls on this alias will be wrapped in lock/unlock instructions for the same lock as the list. This avoids data-races, while allowing direct access to the representation of the list. Note that internal accesses do not need to grab the lock, because the internal views of the object is still **subordinate**.

Only the dominator may "declassify" a subordinate object into a dominator. Since a subordinate object does not know the mode of its dominator, it cannot declassify itself.

5. Summary

The main insight towards previous work that we have gotten from developing KAPPA is that many systems can be closely approximated by thinking about the flow of (read/write) permissions: ownership confines the flow of permissions to some aggregate, immutability removes all the write permissions, uniqueness prohibits duplication of permissions, external uniqueness allows duplication permissions within some aggregate, fractional permissions allows trading write permissions for several read permissions, etc.

Kappa works at the granularity of traits, but each trait conceptually holds a set of read or write permissions to the fields of the trait. Checking interference and controlling the flow of permissions on a higher level of abstraction reduces the annotation overhead and lets us mimic the permission structure of existing systems for alias control by using different combinations of capabilities and modes.

We are standing on the shoulders of giants (*e.g.,* [1–7, 14, 15, 17–22]). With an implementation underway and with many interesting extensions on the horizon, KAPPA will provide a unified system that examines and evaluates how techniques for alias management and different flavours concurrency control can work together in a way that is both versatile and powerful.

Acknowledgments

We thank the anonymous reviewers for their helpful comments and suggestions which improved the paper.

References

[1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *ACM SIGPLAN Notices*, volume 37, pages 311–330. ACM, 2002.

[2] R. L. Bocchino Jr, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. *ACM Sigplan Notices*, 44(10):97–116, 2009.

[3] C. Boyapati. *SafeJava: a Unified Type System for Safe Programming*. PhD thesis, Massachusetts Institute of Technology, 2004.

[4] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *ACM SIGPLAN Notices*, volume 36, pages 56–69. ACM, 2001.

[5] J. Boyland. Alias Burying: Unique Variables without Destructive Reads. *Software: Practice and Experience*, 31(6):533–553, 2001.

[6] J. Boyland. Checking Interference with Fractional Permissions. In R. Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer, 2003. ISBN 3-540-40325-6.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In J. L. Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2001. ISBN 3-540-42206-4.

[8] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. Johnsen, K. Pun, S. Tarifa, T. Wrigstad, and A. Yang. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In M. Bernardo and E. B. Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *LNCS*, pages 1–56. Springer International Publishing, 2015.

[9] S. Brandauer, D. Clarke, and T. Wrigstad. Disjointness Domains for Fine-Grained Aliasing. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 898–916, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5.

[10] E. Castegren and T. Wrigstad. Capable: Capabilities for Scalability. In *Proceedings of the International Workshop on Aliasing Confinement and Ownership (IWACO)*, 2014.

[11] E. Castegren and T. Wrigstad. LOLCAT: Relaxed Linear References for Lock-Free Programming. Technical Report, 2016-013, Uppsala University.

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

[13] D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003. ISBN 3-540-40531-3.

[14] R. DeLine and M. Fähndrich. The Fugue Protocol Checker: Is Your Software Baroque? Technical report, MSR-TR-2004-07, Microsoft Research, 2004.

[15] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *ECOOP 2001—Object-Oriented Programming*, pages 130–149. Springer, 2001.

[16] K. Fernandez-Reyes, D. Clarke, and D. S. McCain. Part: An asynchronous parallel abstraction for speculative pipeline computations. In A. Lluch-Lafuente and J. Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2016. ISBN 978-3-319-39518-0. doi: 10.1007/978-3-319-39519-7_7. URL http://dx.doi.org/10.1007/978-3-319-39519-7_7.

[17] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *ACM SIGPLAN Notices*, volume 26, pages 271–285. ACM, 1991.

[18] Y. Lu, J. Potter, and J. Xue. Ownership Downgrading for Ownership Types. In Z. Hu, editor, *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, volume 5904 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2009. ISBN 978-3-642-10671-2.

[19] F. Militão, J. Aldrich, and L. Caires. Aliasing Control with View-Based Typestate. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, page 7. ACM, 2010.

[20] N. H. Minsky. Towards Alias-Free Pointers. In P. Cointe, editor, *ECOOP'96 - Object-Oriented Programming, 10th European Conference, Linz, Austria, July 8-12, 1996, Proceedings*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer, 1996. ISBN 3-540-61439-7.

[21] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, 1997.

[22] J. Vitek and B. Bokowski. Confined Types in Java. *Software: Practice and Experience*, 31(6):507–532, 2001.

[23] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. *ECOOP 2009 – Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, chapter Loci: Simple Thread-Locality for Java, pages 445–469. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03013-0.