Aliasing control with view-based typestate

Jonathan Aldrich[1] Filipe Militão[1,2] Luís Caires[2]

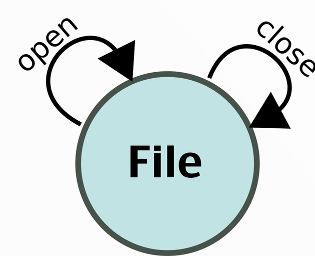
[1]Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

[2]Faculdade de Ciências e Tecnologia - Universidade Nova de Lisboa, Lisboa, Portugal

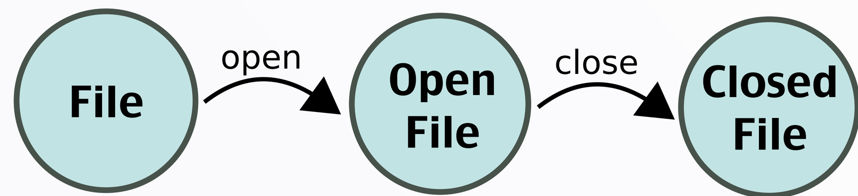
Introduction

Traditional type systems model state implicitly.

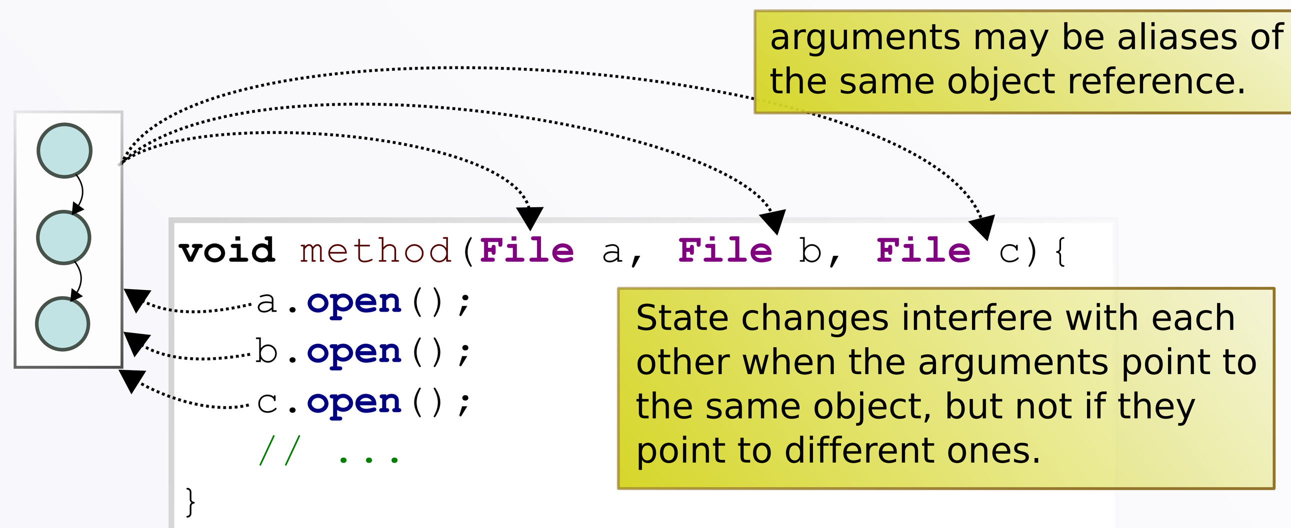
For instance, in a `File` class, although it only makes sense for `close` to be available after a call to `open` they are commonly merged together in the same *class* instead of distinguishing the *states*.



Typestate systems (such as Plural [1]) model state explicitly.



However, this creates the issue of tracking state changes across possibly *aliased* object references.




```
void method(File a, File b, File c){
    a.open();
    b.open();
    c.open();
    // ...
}
```

How can we statically enforce the correct use of state in the presence of alias?

Current solutions restrain access to the aliased reference by using a fixed set of permissions, such as `unique` denoting that there is only one pointer to the object.

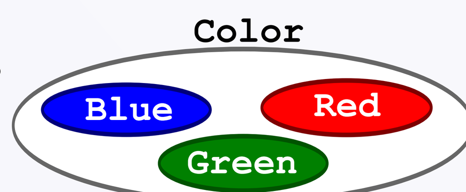
Goal

We hope to make aliasing more explicit, flexible and generic by allowing the programmer to create any number of *views* on an object that more closely model the designer's intent on how a reference should be safely shared/aliased.

View Typestate (based on the PLAID Language)

Views are not fixed: they can be created as needed.

For instance, a `Color` class can define three non overlapping views that allow for independent use of its fields so that no interferences can occur.



Related Work

[1] K. Bierhoff, J. Aldrich. ***Modular Typestate Checking of Aliased Objects.*** OOPSLA 2007.

[2] J. Boyland. ***Checking Interference with Fractional Permissions.*** SAS 2003.

[3] J. C. Reynolds. ***Separation logic: A logic for shared mutable data structures.*** LICS, 2002.

[4] F. Damiani, et al. ***A type safe state abstraction for coordination in Java-like languages.*** Acta Inf, 2008.

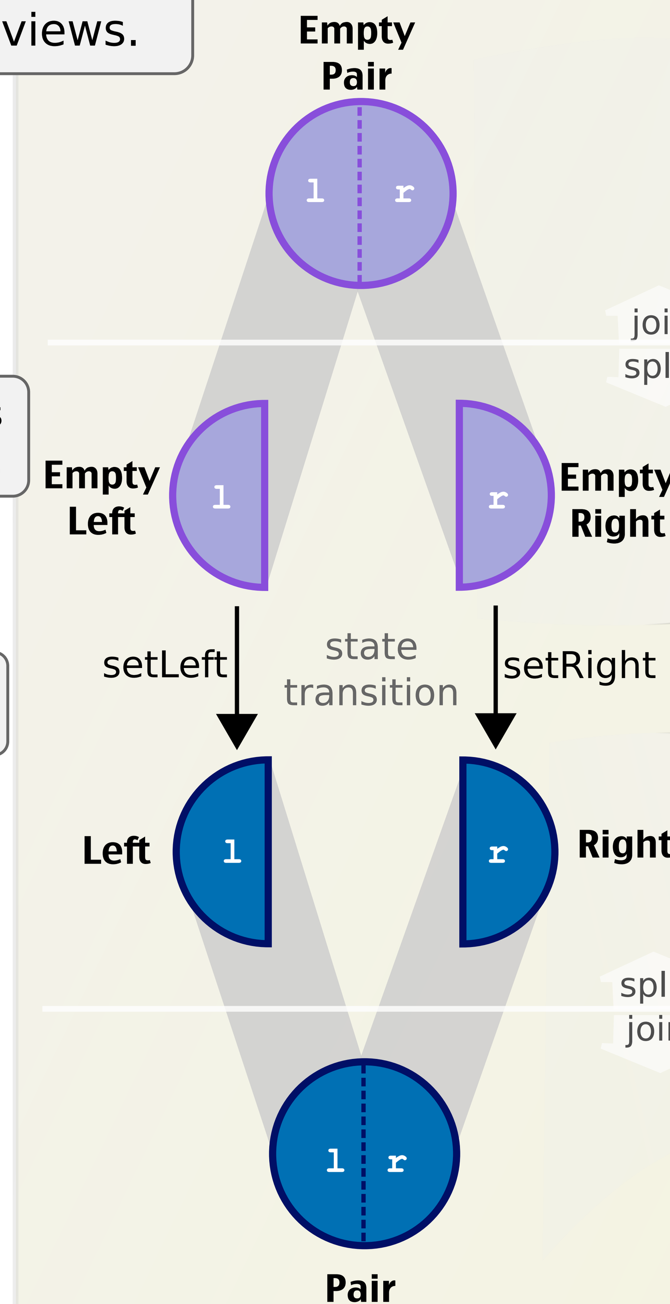
Carnegie Mellon · FCT Faculdade de Ciências e Tecnologia Universidade Nova de Lisboa · PLAID

Pair Example (non overlapping and non interfering)

```
class Pair{
   L l; R r;

   none init(R>>none x, L>>none y) [EmptyPair>>Pair] {
      this.setLeft(y);
      this.setRight(x);
   }

   none setLeft (L>>none x) [EmptyLeft >>Left ] { this.l = x }
   none setRight(R>>none x) [EmptyRight>>Right] { this.r = x }

   // initialized Pair

   int sumSomeR(R>>R x) [Left>>Left] {
      return x.getValue() + this.l.getValue();
   }

   none outsideRight(Right>>Right x) [Left>>Left] {
      //...
   }
   none pair-method() [Pair>>Pair] {
      this.sumSomeR(this.b);
      this.outsideRight(this);
   }
}
...
```

The type system can independently track the initialization state of the *left* and *right* fields through separate views.

- receiver **pre** state/view
- receiver **post** state/view
- the empty (void) permission
- The permission to `L` will be **taken**. Thus afterwards only the empty permission (`none`) remains.
- The permission to `R` will be **borrowed** and fully returned at the end of the method.
- `this.r` is unreachable inside `sumSomeR` (it is in a different view, not in `Left`)
- `this` is automatically split in the two different views: `Left` and `Right` so both this and the previous call are legal




```
...
view EmptyPair { none l; none r; } of Pair

view EmptyLeft  { none l; } of EmptyPair
view EmptyRight { none r; } of EmptyPair

view Left  { L l; } of Pair
view Right { R r; } of Pair
```

views define a slice of a type, that is, a subset of a "larger" type (therefore, all other fields are unreacheable in that view). In future work we will explore more fine grained and flexible **view declarations**, with the possibility of some inference.

```
EmptyPair = EmptyLeft * EmptyRight
```

a **view equation** specifies how an alias is allowed by splitting into separate views (similar to the separation operator in Separation Logic [3]) or how the permission can be recovered by joining the slices of that type.

```
Pair = Left * Right

let z = new Pair() in
  let r = new R() in
    let l = new L() in
      z.init(l,r);
      z.pair-method()
```

initial expression

Iterator Example (overlapping, but non interfering, unbounded sharing)

Iterator Problem: changing a collection during iteration causes undesirable interferences, we want to detect this statically (in Java, a **ConcurrentModificationException** is used to detect the problem but only at run-time, not at compile-time)

```
class Iterator{
   // pretend there is always a next...
   Object next() [Iterator>>Iterator] { ... }
}

class Collection {
   none add(Object>>none o) [Collection>>Collection] { ... }
   int size() [UnderIteration>>UnderIteration] { ... }
}

Collection = UnderIteration * Iterable
? Iterable = ? !Iterable * ? !Iterable
```

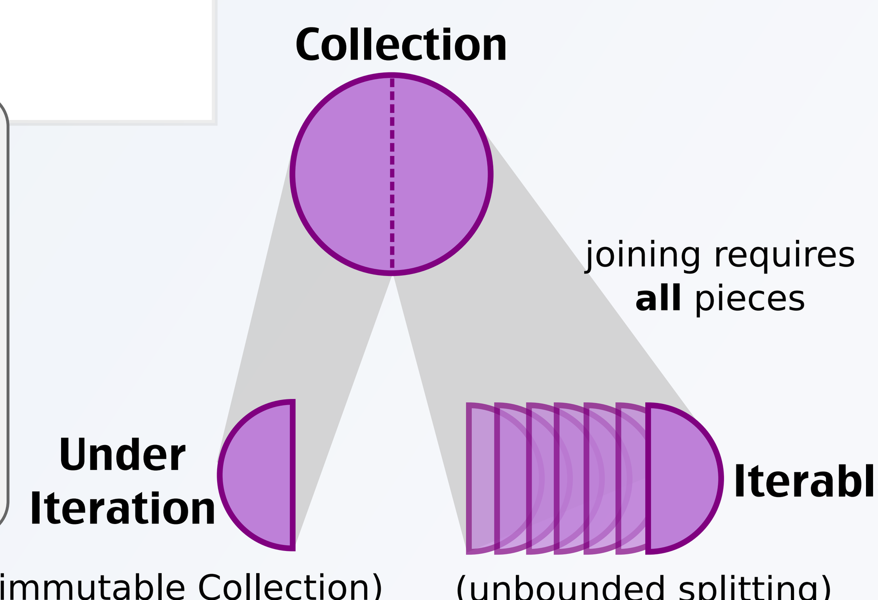
Modifiying a `Collection` is only possible when *all* `Iterable` slices have been collected, otherwise the `Collection` is in an immutable `UnderIteration` view.

`size` does not modify the collection

We employ a simplification of the idea of fractions (from Fractional Permissions [2]) to count the number of existing `Iterable` slices.

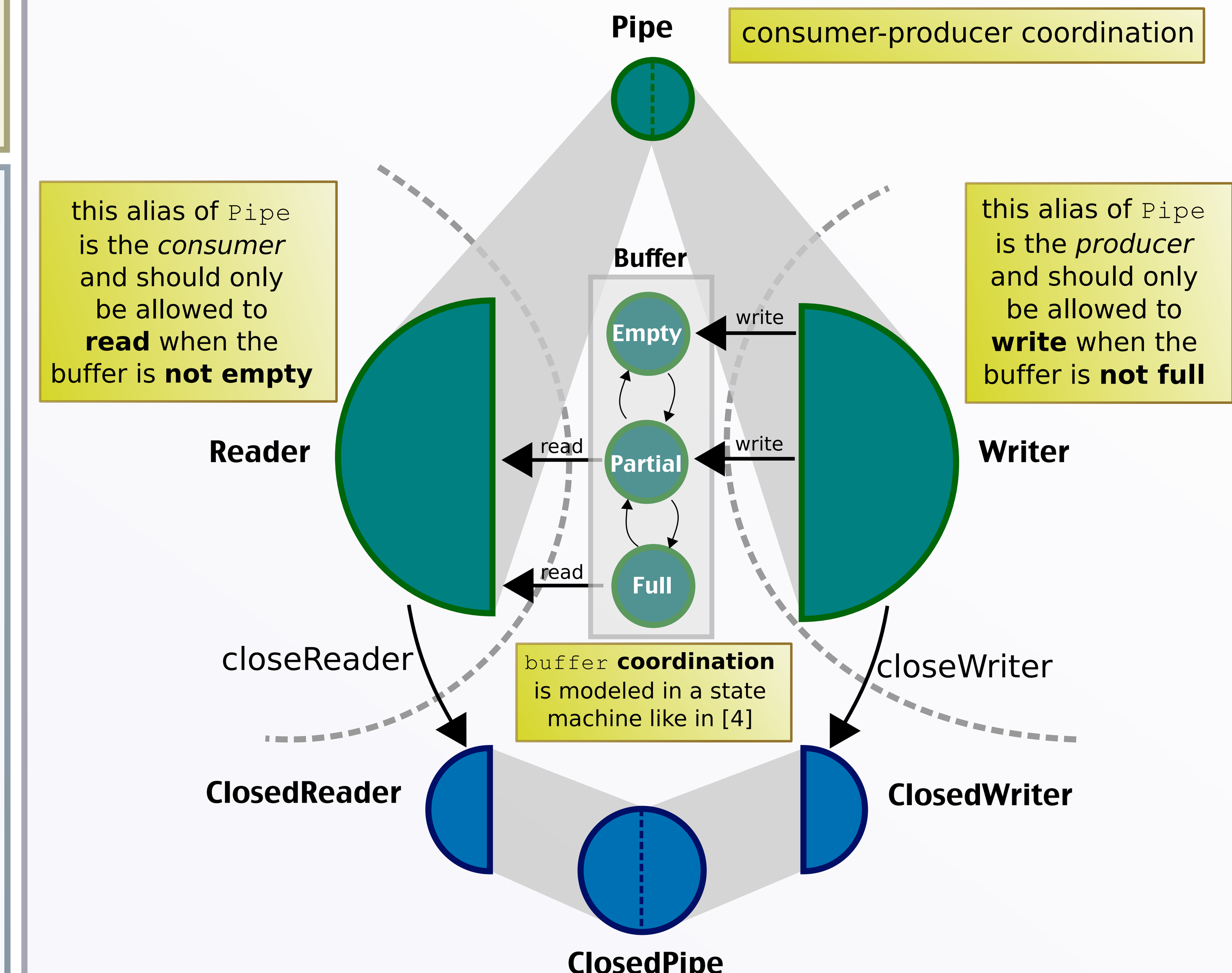
! represents a fixed fraction of 1/2 value. So in the begining we have: `Iterable = !Iterable * !Iterable`

? is used to represent some fixed number of !'s (0 or more), therefore for ? = !! this second view equation allows for: `!!Iterable = !!!Iterable * !!!Iterable`



(work in progress...)

Future Work: overlapping and interfering sharing with coordination



Research Questions

How to coordinate and typecheck sharing of overlapping data? **(including other types of coordination, such as multiple readers with a single writer, etc.)**

Does knowing how each view uses its slice of a class help in checking race conditions, incorrect coordination (deadlocks), and lock/atomic blocks? (for instance, merging views that were used by different threads can expose non-isolation, etc.)