Miniboxing

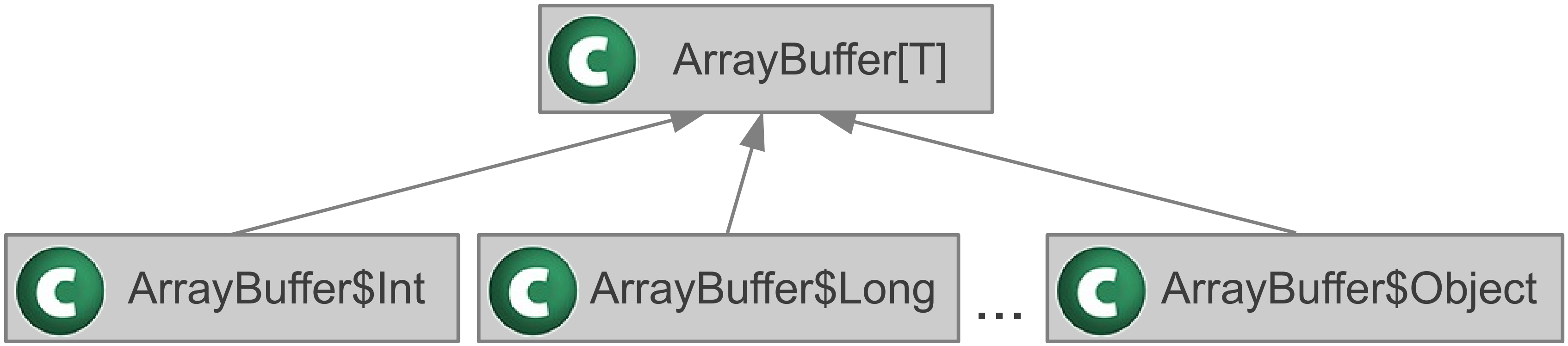
Runtime Specialization on the JVM

Under the hood presentation
LAMP, EPFL, 6th of March 2013

Vlad Ureche
vlad.ureche@epfl.ch

Generic Code

- enables reuse and safety
- value types - pointers (Object) vs stack values
- common representation - boxing? NO



We are looking for specialization

Outline

- Generic code
  - Problem
  - State of the art
- Miniboxing
- Evolving miniboxing
- Evaluation

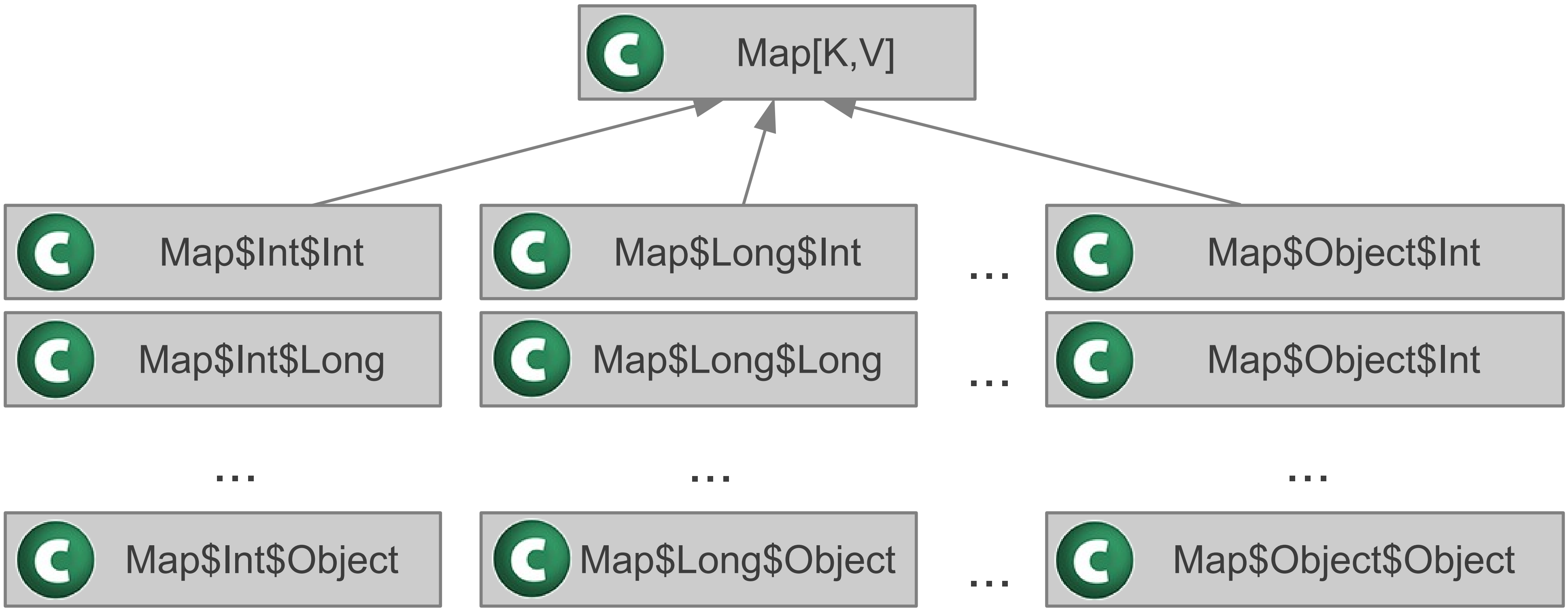
State of the Art

- dot net
  - runtime specialization, in the virtual machine
  - doesn't apply to Java bytecode - erasure
- c++ templates
  - expand at compile time, as needed
  - duplicate instantiations open the door to incompatibilities
- pizza
  - runtime specialization via classloader
  - complex changes, reflection, slow

Hard problem

State of the art - Specialization

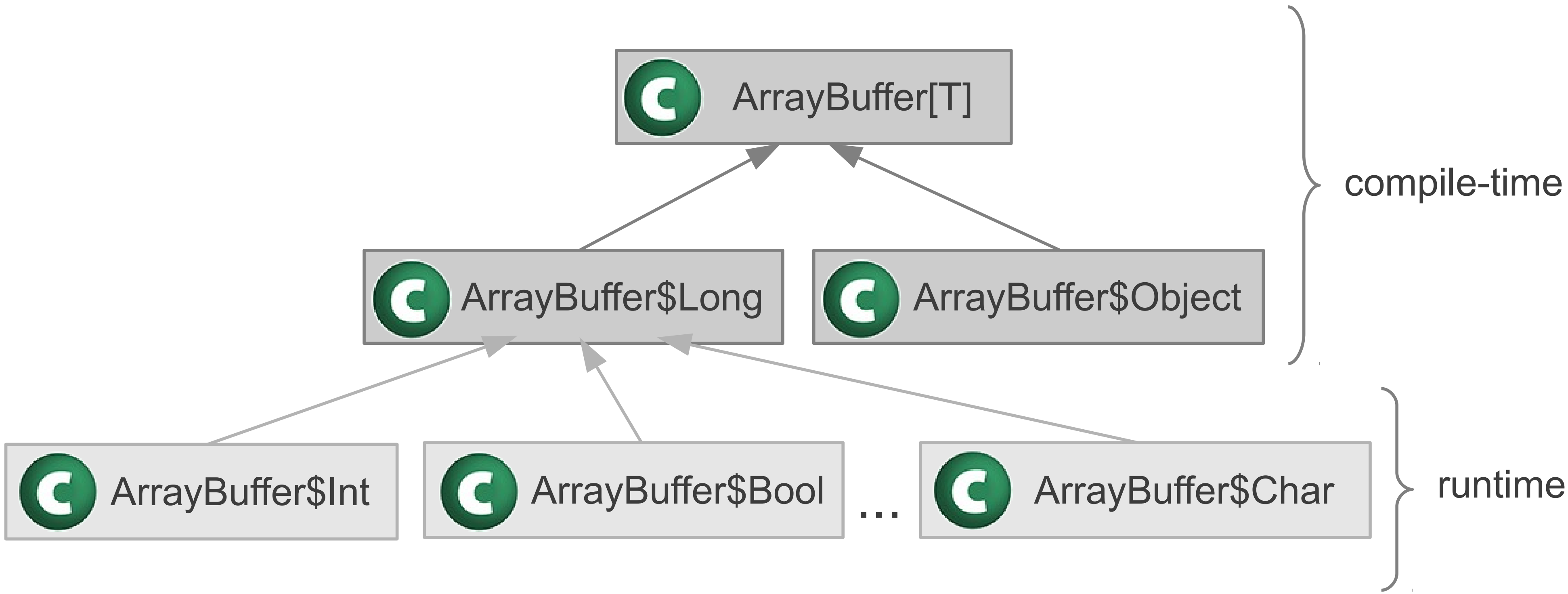
- static expansion, in the library



81 classes is too much - we need a lighter approach

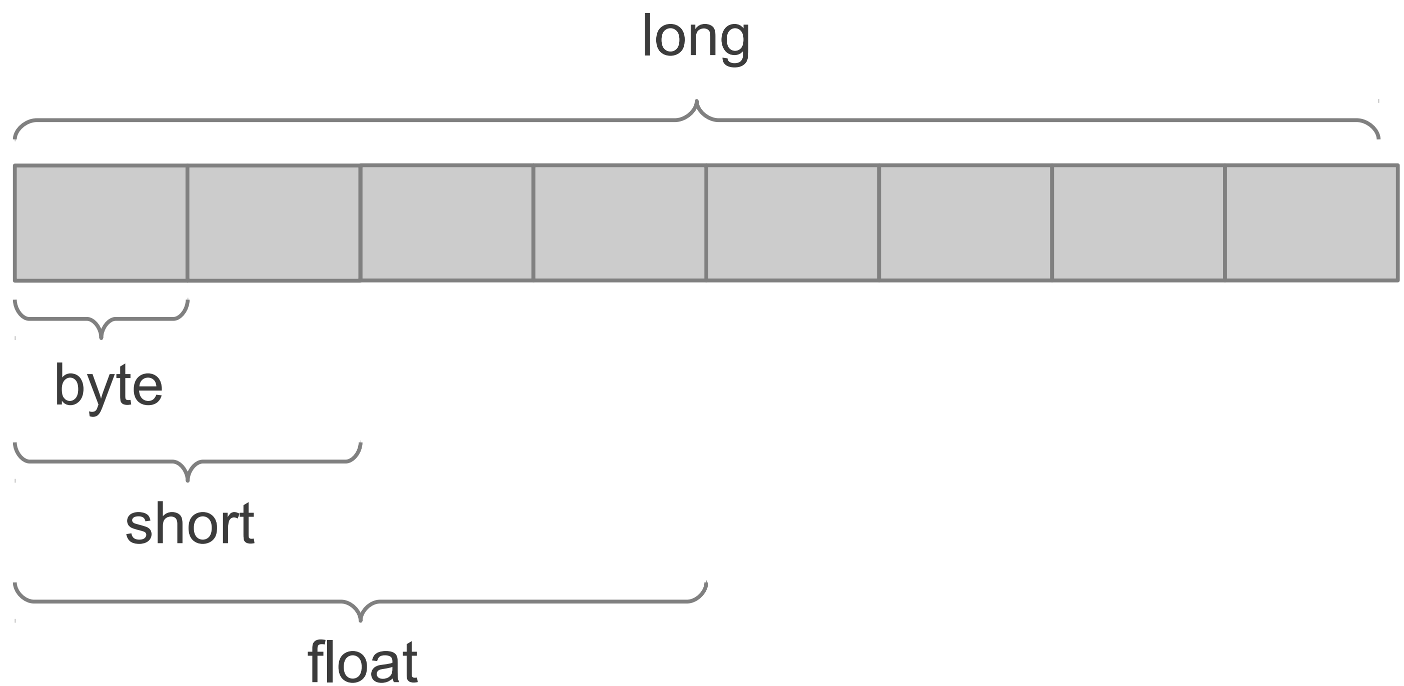
Miniboxing?

- dot net-like runtime specialization
  - two stages: compile-time and runtime



Code is also faster than specialization

Stack Insight

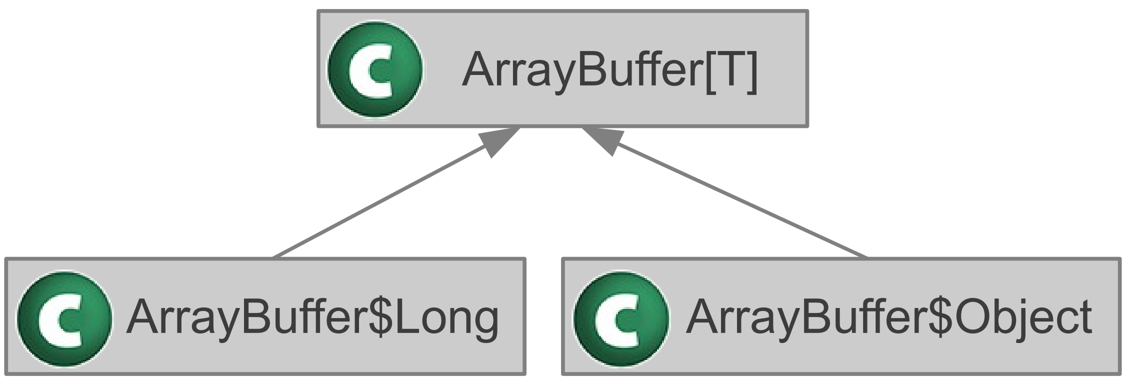


- Only applied to stack values
- We'll look at arrays separately.

One **large** value type is enough

Miniboxing

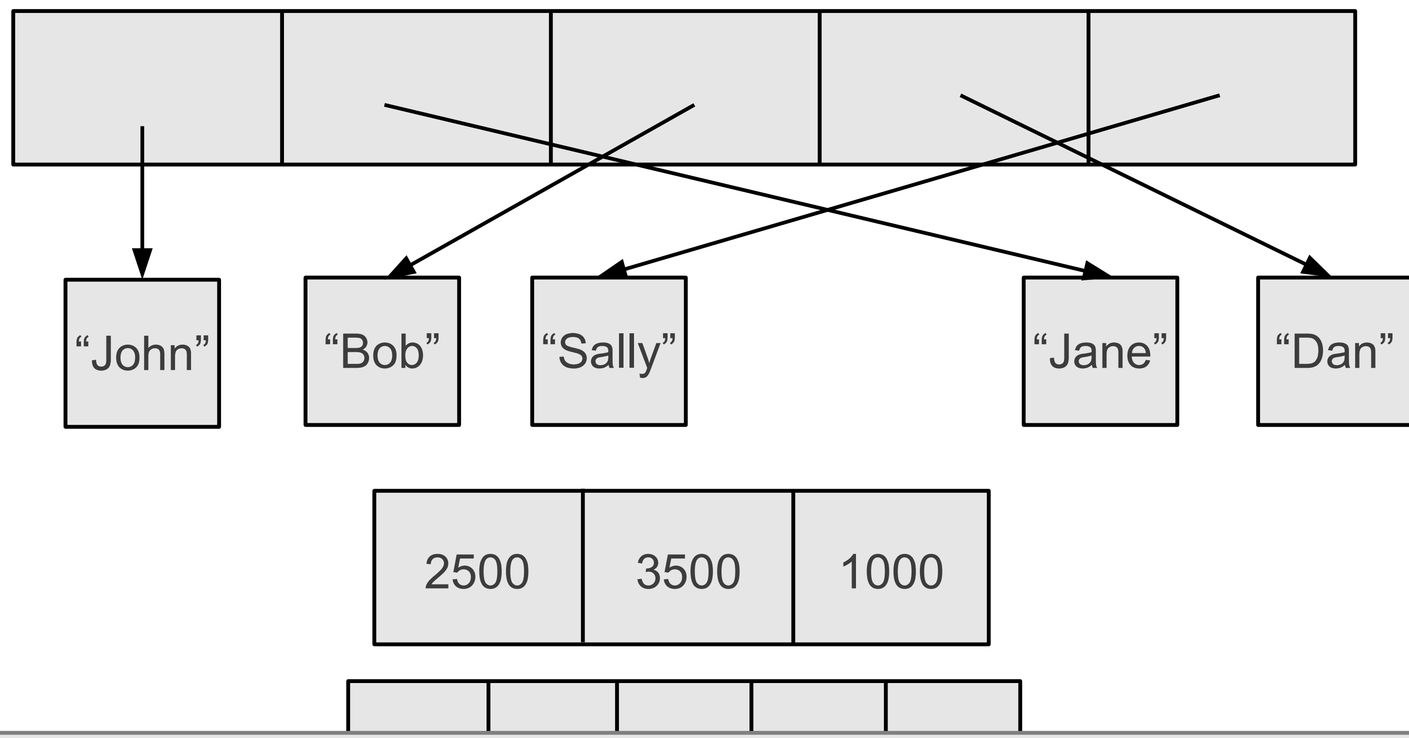
- object-specialized class
- long-specialized class
- let's leave runtime aside for now



Even this is tricky. Why?

ArrayBuffer

- Uses an underlying array
  - Array[String], Array[Int], Array[Float]



Access the array through a library

Library Support

- Executing code on value types (.hashCode...)
- Interfacing with arrays

```
array(i)
```

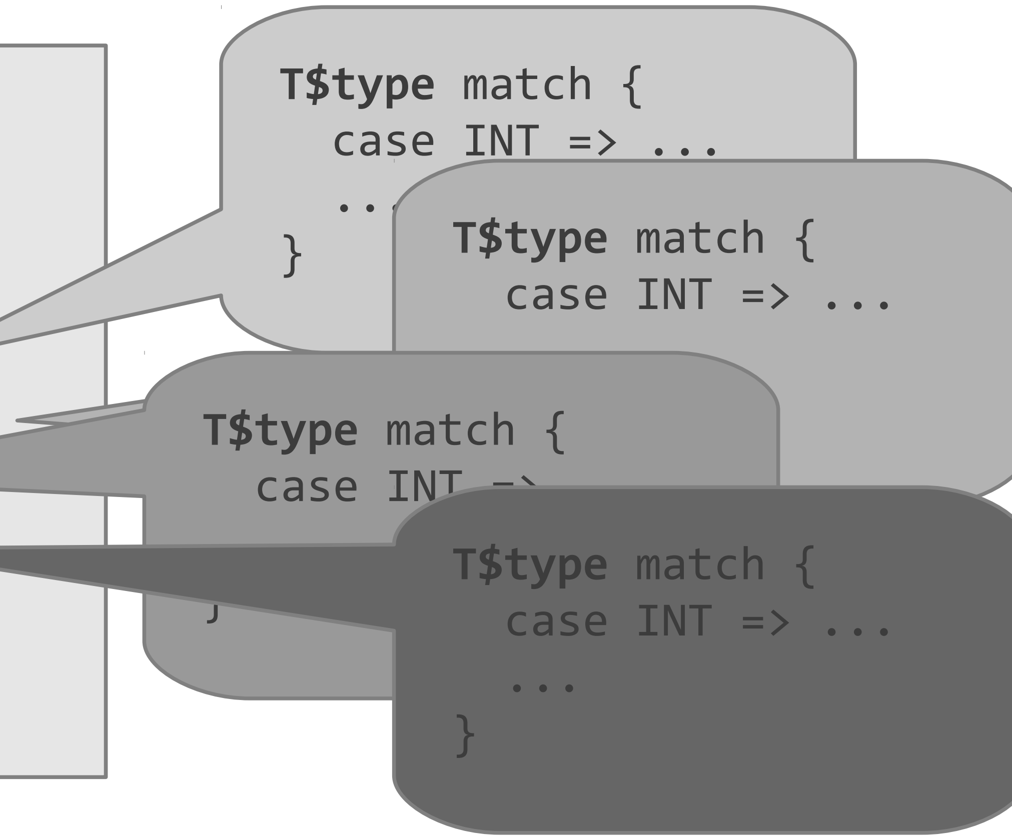
```
T$type match {
  case INT => array.asInstanceOf[Array[Int]](i).toLong
  ...
}
```

Surprisingly, casting and converting to long is fast

Library Support

- `ArrayBuffer.reverse()`

```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```



Can the JVM optimize such code?

Java Virtual Machine Support

- We confuse the JVM heuristics by inlining manually
- The JVM will do its best
  - For one value of T$Type (one type)
    - hoist the switch outside the loop (loop unswitching)
    - get rid of multiple casts (common subexpression elim)
    - get rid of muliple bounds checks (check elimination)
    - **good speed**
  - For more types… there are tricks
    - **no good solution to cover all cases**

Can we lift the switch ourselves?

Dispatching

- Systems approach: `T$type`
  - byte value
  - encodes the type of a parameter (`INT`, `...`)
  - `switch` to do the right operation
- Object-oriented approach
  - common operations object (Dispatcher)
  - specialized instances for all types

Dispatching

- `T$Dispatcher: Dispatcher`
  - `IntDispatcher, LongDispatcher, ...`
- Common operators

```
array(i)
```

```
T$Dispatcher.array_get(array, i)
```

So far so good

Dispatching

- `ArrayBuffer.reverse()`

```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val other = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(other)
    array(index) = tmp2
    array(other) = tmp1
    index += 1
  }
}
```

T$Dispatcher.array_get

T$Dispatcher.array_get

T$Dispatcher.array_update

T$Dispatcher.array_update

We committed to the data representaion before

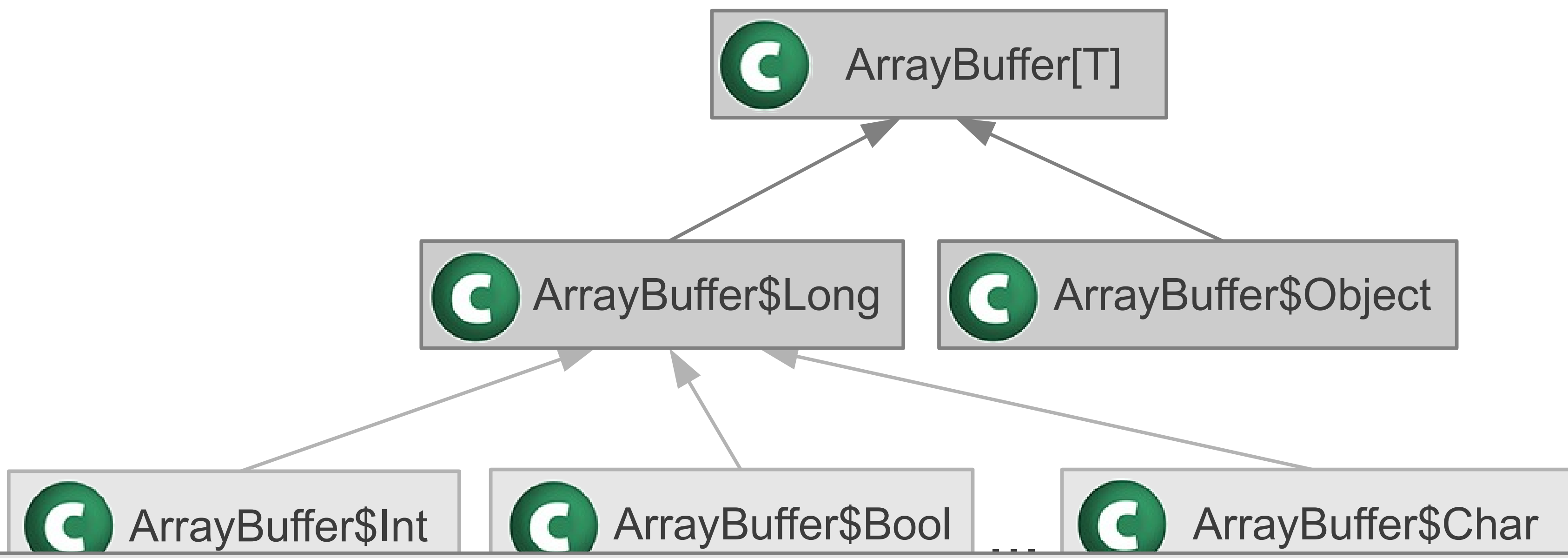
Dispatching Performance

- One or two dispatchers in a callsite
  - as fast as manually specialized code
- Three+ dispatchers
  - call to `array_get` becomes **megamorphic**
  - it is not inlined anymore
  - no optimization in the loop
  - awful results, in some cases slower than generic

Can't use such a solution. Or can we?

Runtime Specialization

- In practice, dispatcher set per INSTANCE
- Semantically - dispatcher per SPECIALIZATION
- Remember the ArrayBuffer diagram?



We can statically set the dispatcher

Runtime Specialization

- `ArrayBuffer$Int`
  - has `IntDispatcher` set statically
- Operations can be inlined
  - since `IntDispatcher` is final
  - calls never become megamorphic
  - optimizations can be done inside the loop

Class modification and loading cost?

Runtime Specialization

- One-time non-negligible cost
  - load `...$Long` bytecode
  - modify it
  - build a `Class[_]` object
  - instantiate it
- Apply the cost to the factory
  - instead of the individual class
  - turns out the cost amortizes well (10-50 uses)

Class modification and loading cost?

Numbers Preview

- `ArrayBuffer.reverse()`

```
                       ideal :     0.85883 ms +/-   0.05687
     miniboxed standard mono :     0.62966 ms +/-   0.21116
     miniboxed standard mega :     4.21423 ms +/-   1.15425
     miniboxed standard w/cl :     0.87526 ms +/-   0.05992
     miniboxed dispatch mono :     0.75230 ms +/-   0.21551
     miniboxed dispatch mega :     4.90191 ms +/-   1.52366
miniboxed dispatch w/cl mono :     0.77984 ms +/-   0.14872
miniboxed dispatch w/cl mega :     0.80932 ms +/-   0.14671
            specialized mono :     0.75813 ms +/-   0.16000
            specialized mega :     0.61374 ms +/-   0.20144
                     generic :     9.76250 ms +/-   1.55297
```

Conclusions

- Yes, this can be done
  - There's no silver bullet
  - We just build around limitations
  - And need to make things as robust as possible
- Contributions
  - Miniboxing itself, in an open-world assumption
  - Exploration of the implementation space
    - Dispatchers
    - Runtime specialization
  - Solution that works in practice

Thank you!

github.com/miniboxing

Vlad Ureche
vlad.ureche@epfl.ch

Compatibility

- a common set of operations
  - not only the generic operations
    - we want values too