Miniboxing

Load-time Specialization on the JVM

OOPSLA, 29th of October 2013

Vlad Ureche
Cristian Talau
Martin Odersky

We all like generics

We all like generics

a trivial example

We all like generics

a trivial example

```
def identity[T](t: T): T = t
```

We all like generics

a trivial example

```
def identity[T](t: T): T = t
```

- will take any type and
- will return **that same type**

We all like generics

a trivial example

```
def identity[T](t: T): T = t
```

- will take any type and
- will return **that same type**

We all like generics

a trivial example

```
def identity[T](t: T): T = t
```

but under **erasure:**

```
def identity(t: Any): Any = t
```

We all like generics

a trivial example

```
def identity[T](t: T): T = t
```

but under **erasure:**

Any is the top of the Scala type system

```
def identity(t: Any): Any = t
```

We all like generics

a trivial example

```
x = identity(3)
```

We all like generics

a trivial example

```
x = identity(3)
```

under **erasure:**

```
x = unbox(identity(box(3)))
```

We all like generics

but under **erasure**

We all like generics

but under **erasure**

generics execute similarly to **dynamic languages**

We all like generics

but under **erasure**

generics execute similarly to **dynamic languages**

- generic values **lose** their **type information**

We all like generics

but under **erasure**

generics execute similarly to **dynamic languages**

- generic values **lose** their **type information**
- primitives need **boxing**

We all like generics

but under **erasure**

generics execute similarly to **dynamic languages**

- generic values **lose** their **type information**
- primitives need **boxing**
- performance is **affected**

We all like generics

but under **erasure**

generics execute similarly to **dynamic languages**

- generic values **lose** their **type information**
- primitives need **boxing**
- performance is **affected**

Dynamic language VMs use **specialization** to improve performance*

We all like generics

but under **erasure**

generics execute similarly to **dynamic languages**

- generic values **lose** their **type information**
- primitives need **boxing**
- performance is **affected**

Dynamic language VMs use **specialization** to improve performance*

*but the HotSpot JVM **doesn't**

Generics

WE ARE HERE

Specialization

Miniboxing

Performance

Evaluation

Scala has a solution

Scala has a solution

it's called specialization*

[*] Iulian Dragos – PhD thesis, EPFL, 2010

Scala has a solution

it's called specialization*

* Iulian Dragos – PhD thesis, EPFL, 2010

Compile-time (static) transformation

- **duplicates** the original code
- **adapts it** for each primitive type
- **rewrites** programs to use the adapted code

Scala has a solution

it's called specialization[*]

[*] Iulian Dragos – PhD thesis, EPFL, 2010

Compile-time (static) transformation

- **duplicates** the original code
- **adapts it** for each primitive type
- **rewrites** programs to use the adapted code

Adapted code doesn't need to box

Scala has a solution

it's called specialization*

Compile-time (static) transformation

- **duplicates** the original code
- **adapts it** for each primitive type
- **rewrites** programs to use the adapted code

Adapted code doesn't need to box

Performance is regained.

* Iulian Dragos – PhD thesis, EPFL, 2010

Specialization

let's revisit `def identity`

Specialization

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

Specialization

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_V(t: Unit): Unit = t
def identity_Z(t: Boolean): Boolean = t
def identity_B(t: Byte): Byte = t
def identity_C(t: Char): Char = t
def identity_S(t: Short): Short = t
def identity_I(t: Int): Int = t
def identity_J(t: Long): Long = t
def identity_F(t: Float): Float = t
def identity_D(t: Double): Double = t
```

Specialization

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_V(t: Unit): Unit = t
def identity_Z(t: Boolean): Boolean = t
def identity_B(t: Byte): Byte = t
def identity_C(t: Char): Char = t
def identity_S(t: Short): Short = t
def identity_I(t: Int): Int = t
def
def identity_F(t: Float): Float = t
def identity_D(t: Double): Double = t
```

Generates 10 times the original code

Specialization

… it gets even worse

Specialization

... it gets even worse

```
def pack[T1, T2](t1: T1, t2: T2) = ...
```

Specialization

... it gets even worse

```
def pack[T1, T2](t1: T1, t2: T2) = ...
```

```
def pack_VV(t1: Unit, t2: Unit)
def pack_VZ(t1: Unit, t2: Boolean)
def pack_VB(t1: Unit, t2: Byte)
def pack_VC(t1: Unit, t2: Char)
def pack_VS(t1: Unit, t2: Short)
def pack_VI(t1: Unit, t2: Int)
def pack_VJ(t1: Unit, t2: Long)
def pack_VF(t1: Unit, t2: Float)
def pack_VD(t1: Unit, t2: Double)
```

Specialization

… it gets even worse

```
def pack[T1, T2](t1: T1, t2: T2) = ...
```

```
def pack_VV(t1: Unit, t2: Unit)
def pack_VZ(t1: Unit, t2: Boolean)
de
de
def pack_VS(t1: Unit, t2: Short)
def pack_VI(t1: Unit, t2: Int)
def pack_VJ(t1: Unit, t2: Long)
def pack_VF(t1: Unit, t2: Float)
def pack_VD(t1: Unit, t2: Double)
```

10^n, where n is the number of type params

Specialization

… it gets even worse

```
def pack[T1, T2](t1: T1, t2: T2) = ...
```

```
def pack_VV(t1: Unit, t2: Unit)
def pack_VZ(t1: Unit, t2: Boolean)
de
de
de
de
def pack_VJ(t1: Unit, t2: Long)
def pack_VF(t1: Unit, t2: Float)
def pack_VD(t1: Unit, t2: Double)
```

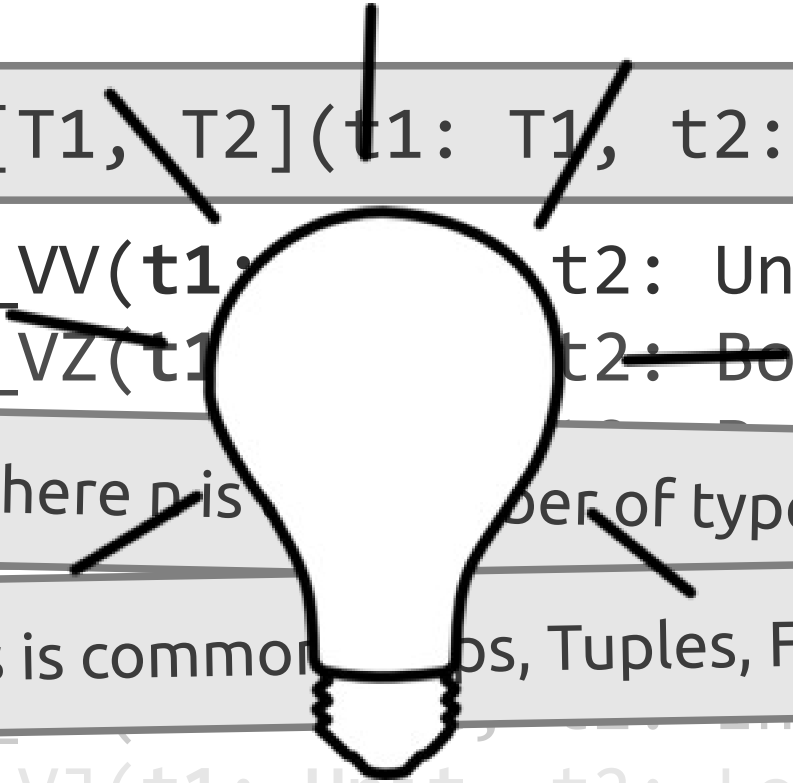
10^n, where n is the number of type params

And this is common: Maps, Tuples, Functions

Specialization

... it gets even worse

```
def pack[T1, T2](t1: T1, t2: T2) = ...
```



```
def pack_VV(t1:          t2: Unit)
def pack_VZ(t1           t2: Boolean)
de
de
de
de                             Int)
def pack_VJ(t1: Unit, t2: Long)
def pack_VF(t1: Unit, t2: Float)
def pack_VD(t1: Unit, t2: Double)
```

10^n, where n is   ber of type params

And this is common   s, Tuples, Functions

Generics

Specialization

Miniboxing

WE ARE HERE

Performance

Evaluation

Miniboxing

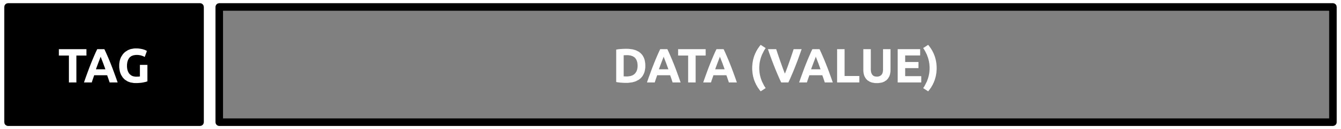
Miniboxing

reduces the variants

Miniboxing

reduces the variants

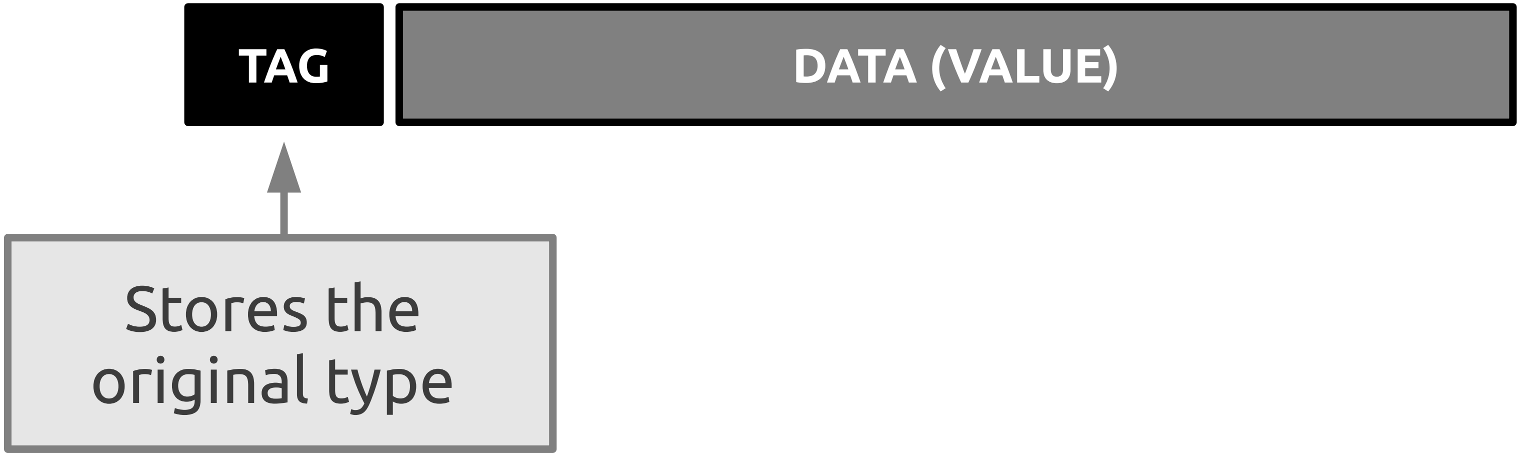
by using something like a **tagged union**



Miniboxing

reduces the variants

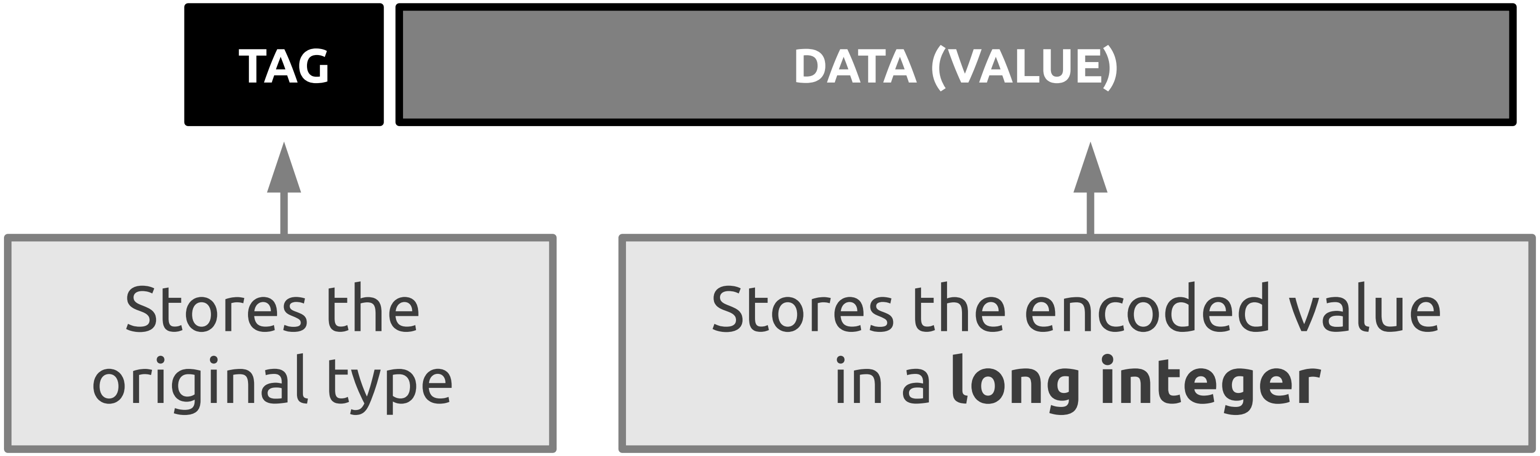
by using something like a **tagged union**



Miniboxing

reduces the variants

by using something like a **tagged union**



TAG

DATA (VALUE)

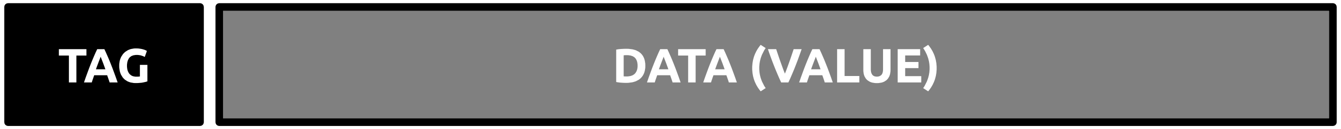
Stores the original type

Stores the encoded value in a **long integer**

Miniboxing

reduces the variants

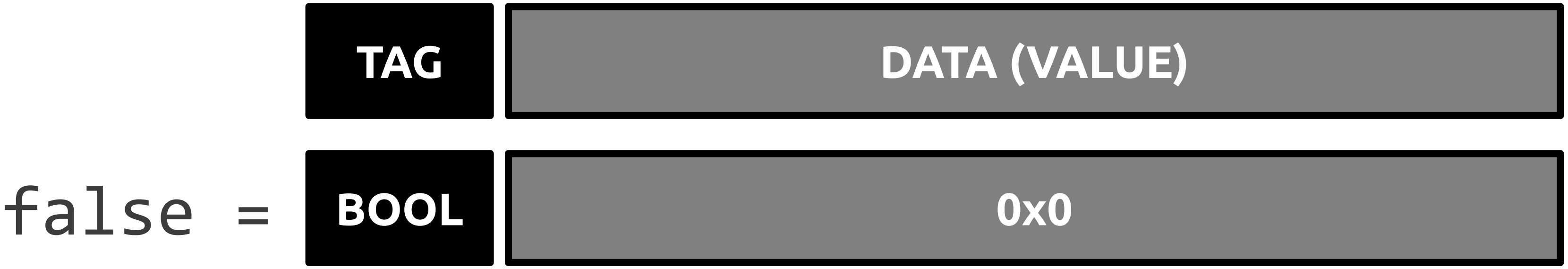
by using something like a **tagged union**



Miniboxing

reduces the variants

by using something like a **tagged union**

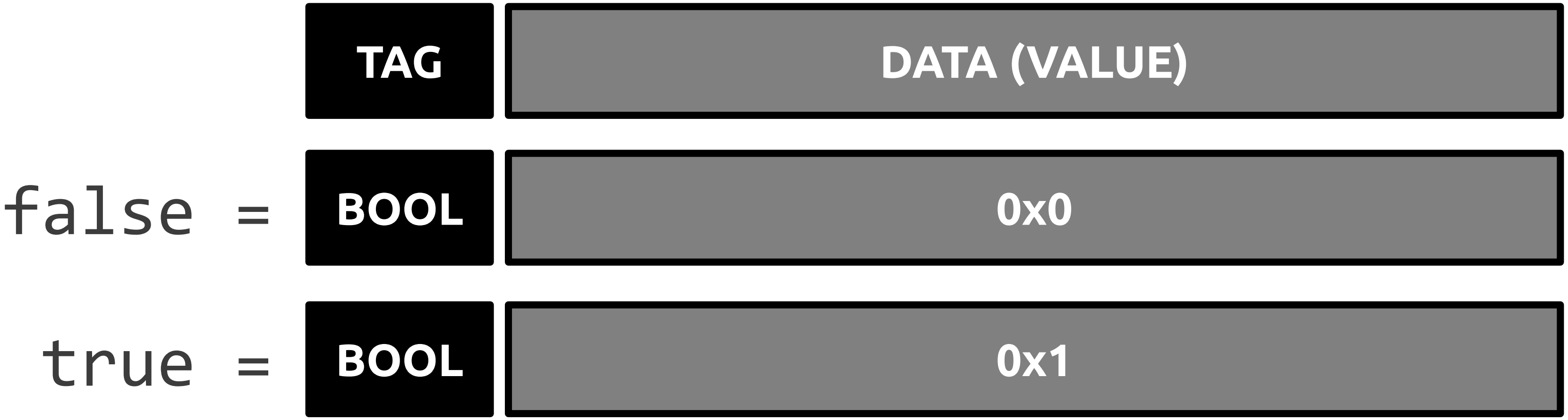


	TAG	DATA (VALUE)
`false =`	BOOL	0x0

Miniboxing

reduces the variants

by using something like a **tagged union**



	TAG	DATA (VALUE)
`false =`	BOOL	0x0
`true =`	BOOL	0x1

Miniboxing

reduces the variants

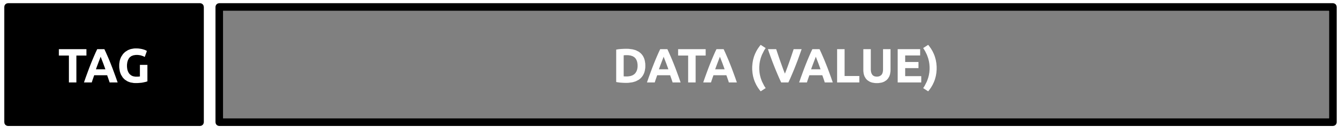
by using something like a **tagged union**

	TAG	DATA (VALUE)
`false =`	BOOL	0x0
`true =`	BOOL	0x1
`42 =`	INT	0x2A

Miniboxing

reduces the variants

by using something like a **tagged union**

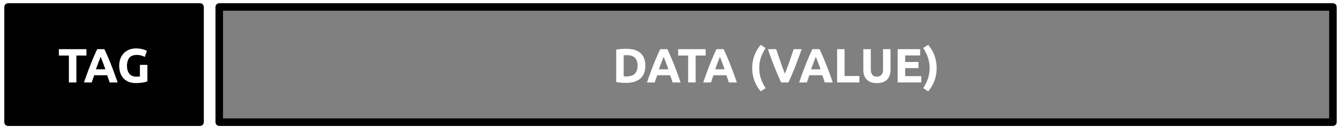


TAG

DATA (VALUE)

Miniboxing
reduces the variants

by using something like a **tagged union**



TAG

DATA (VALUE)

and using the **static type information**

- tags are attached to **code**, not to values

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```

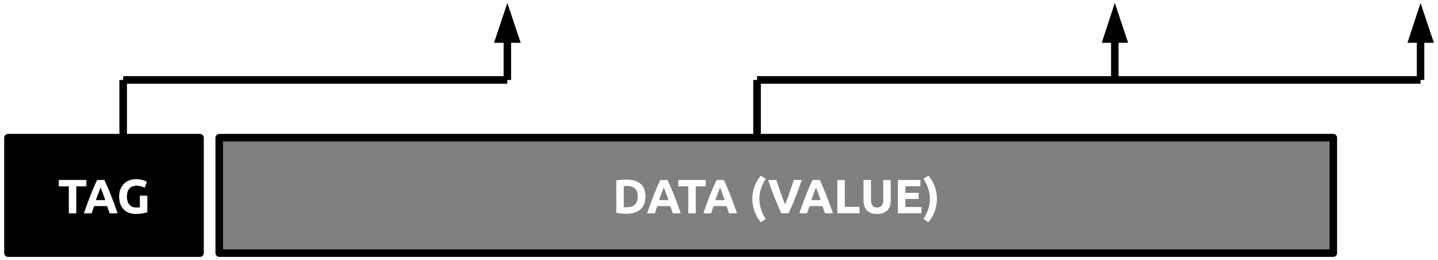
TAG

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```



TAG

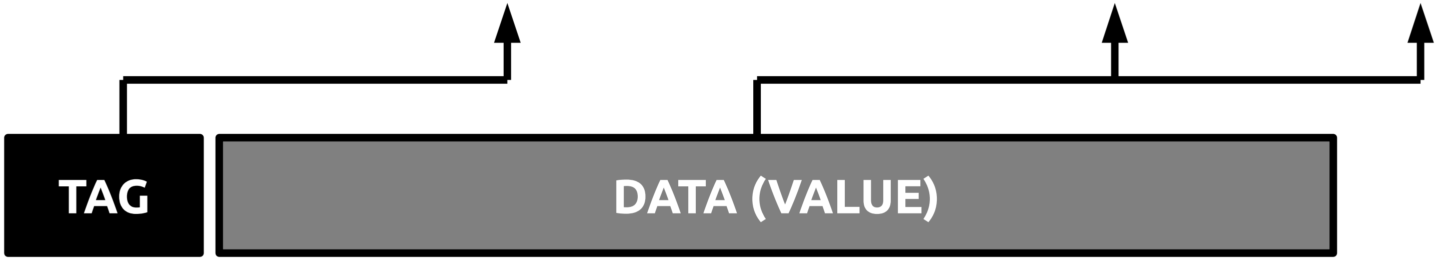
DATA (VALUE)

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```



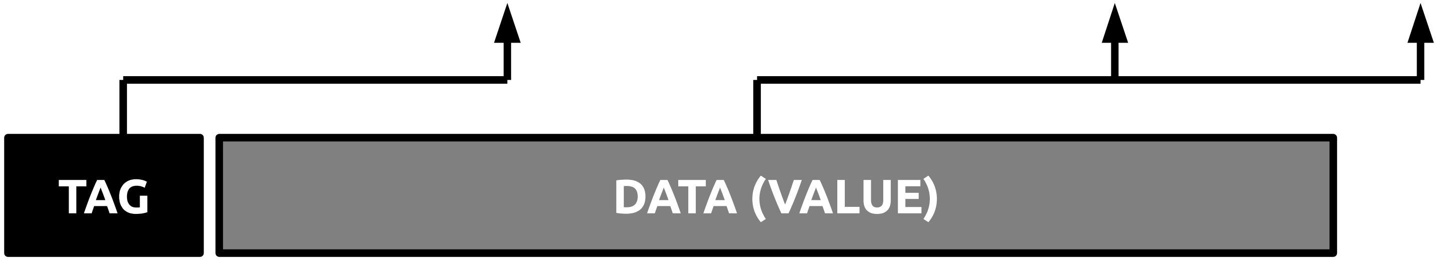
T_tag corresponds to the **type parameter**, instead of the values being passed around.

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```



T_tag corresponds to the **type parameter**, instead of the value [illegible] und.

Tag hoisting

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```

Two variants per type parameter (reference + minibox)

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```

Two variants per type parameter (reference + minibox)

`def pack` will have 4 variants

Miniboxing

let's revisit `def identity`

```
def identity[T](t: T): T = t
```

```
def identity_M(T_tag: Byte, t: Long): Long
```

Two variants per type parameter (reference + minibox)

`def pack` will have 4 variants

Tag hoisting is instrumental in obtaining good performance

Generics

Specialization

Miniboxing

Performance

WE ARE HERE

Evaluation

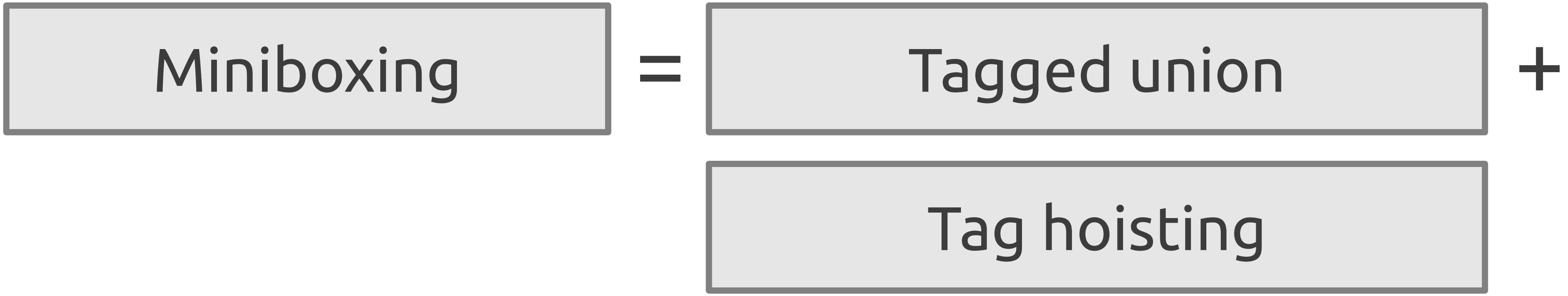
Performance

needs one more ingredient

Miniboxing = Tagged union

Performance

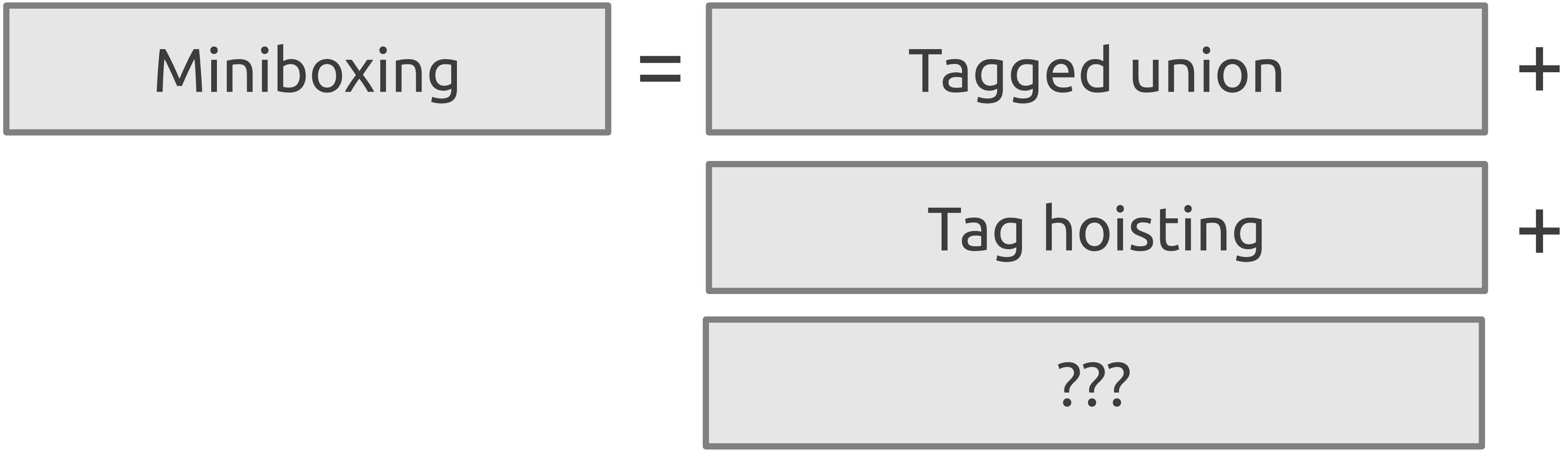
needs one more ingredient



Miniboxing = Tagged union + Tag hoisting

Performance

needs one more ingredient



Miniboxing = Tagged union + Tag hoisting + ???

Performance

needs one more ingredient

Miniboxing = Tagged union + Tag hoisting + ???

Why do we need a **secret ingredient?**

Switching on tags

kills performance

Switching on tags

kills performance

```
def toString(T_tag: Byte,
             value: Long): String =
  T_tag match {
    case UNIT => ...
    case BOOL => ...
    ...
  }
```

Switching on tags

kills performance

```
def toString(T_tag: Byte,
             value: Long): String =
  T_tag match {
    case UNIT => ...
    case BOOL => ...
    ...
  }
```

Even more so for consecutive switches

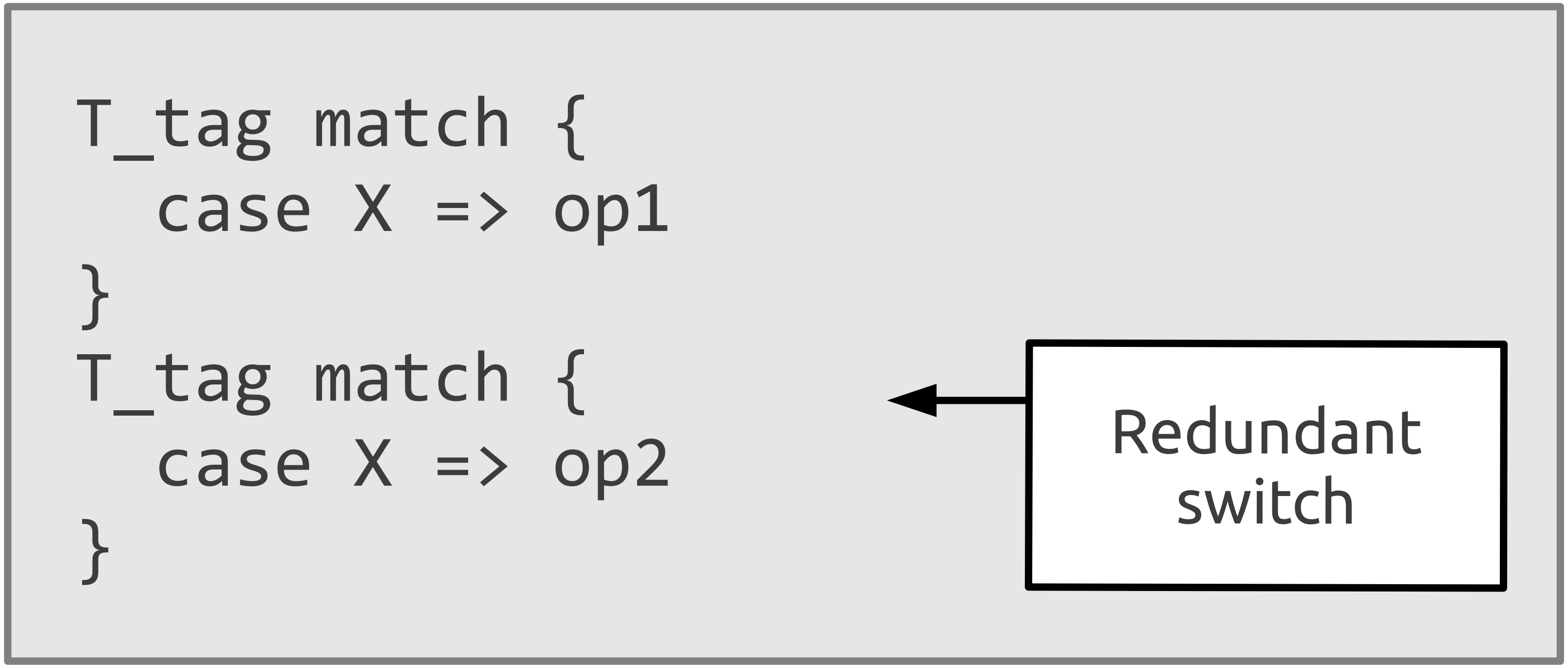
Switching on tags

kills performance

```
T_tag match {
  case X => op1
}
T_tag match {
  case X => op2
}
```

Switching on tags

kills performance

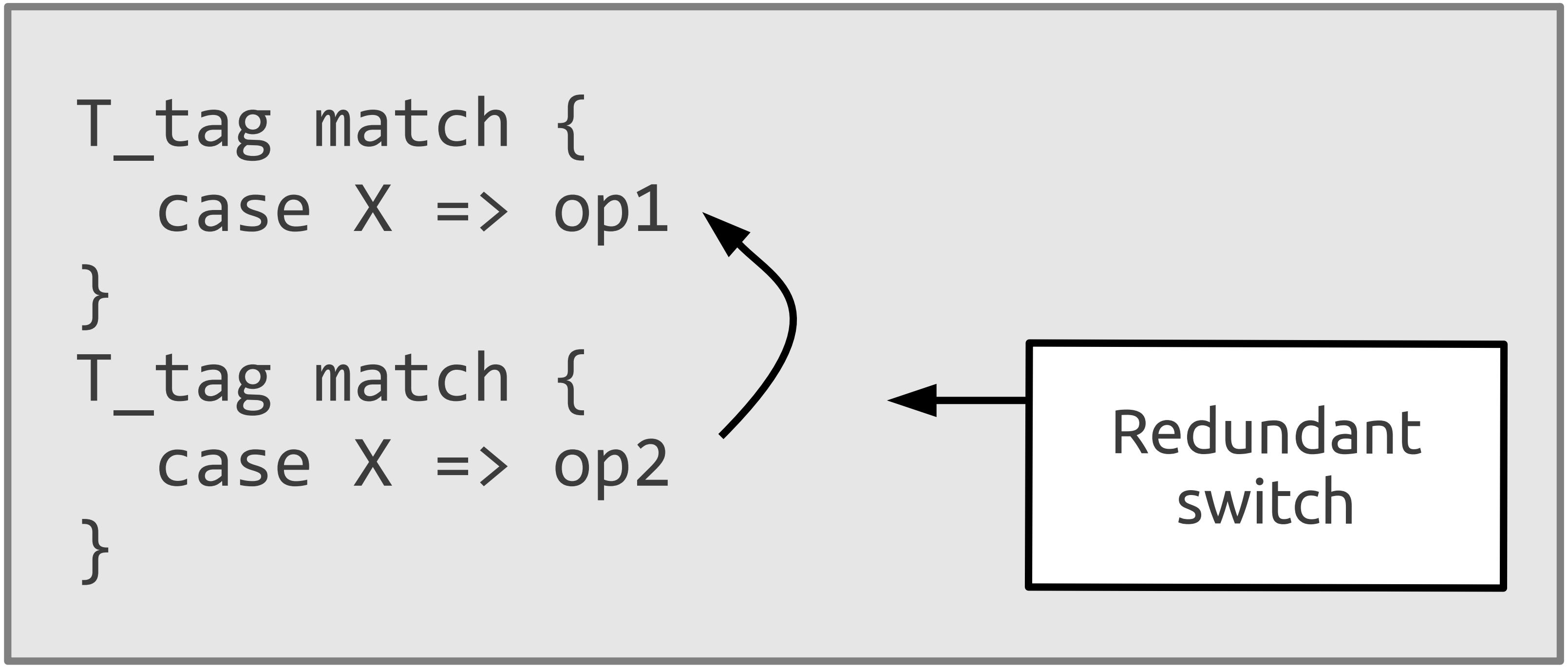


```
T_tag match {
  case X => op1
}
T_tag match {
  case X => op2
}
```

Redundant switch

Switching on tags

kills performance

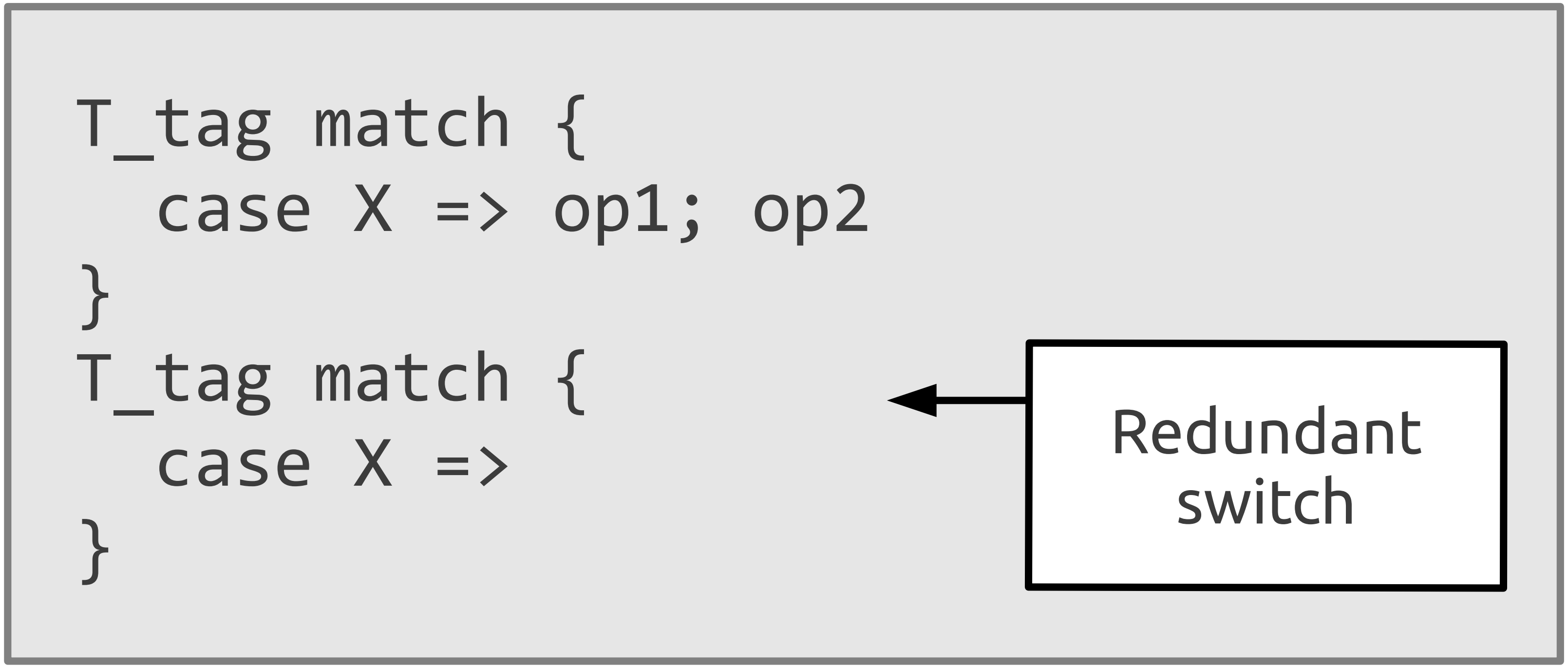


```
T_tag match {
  case X => op1
}
T_tag match {
  case X => op2
}
```

Redundant switch

Switching on tags

kills performance

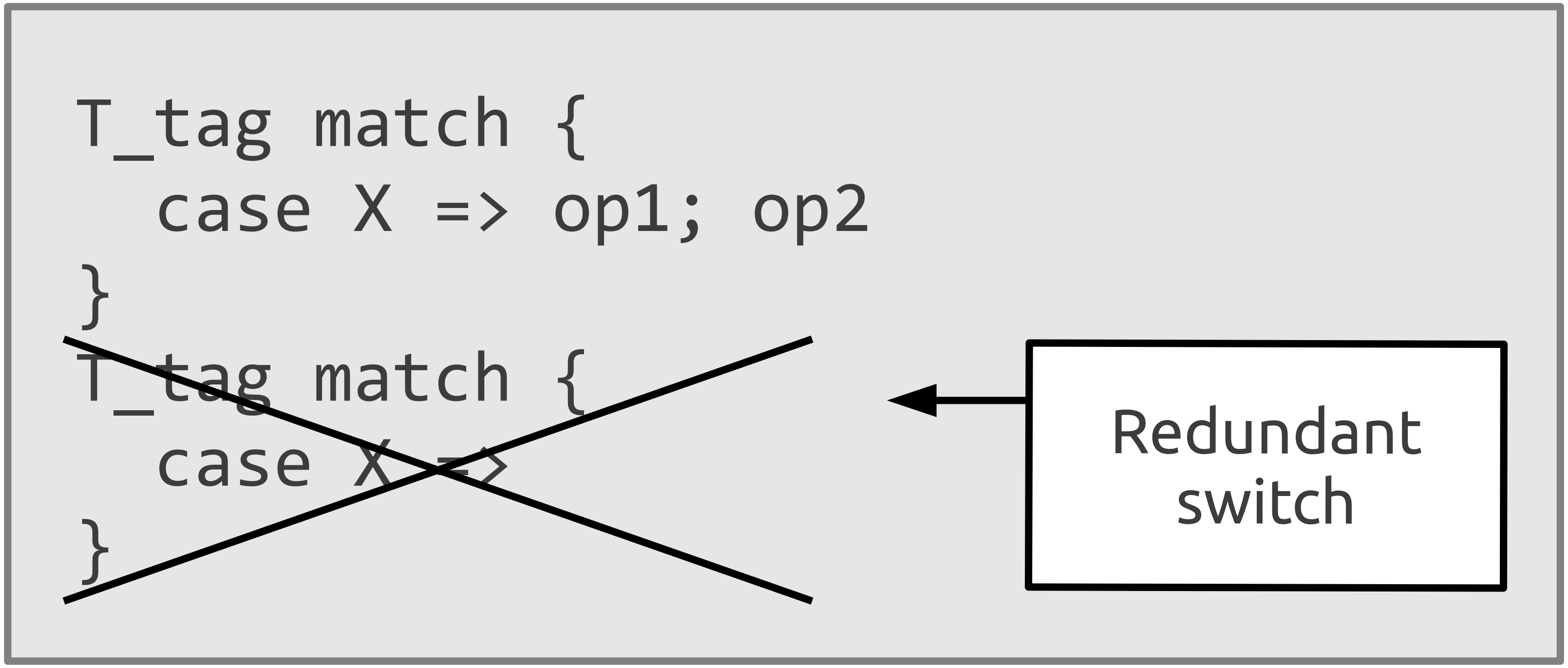


```
T_tag match {
  case X => op1; op2
}
T_tag match {
  case X =>
}
```

Redundant switch

Switching on tags

kills performance



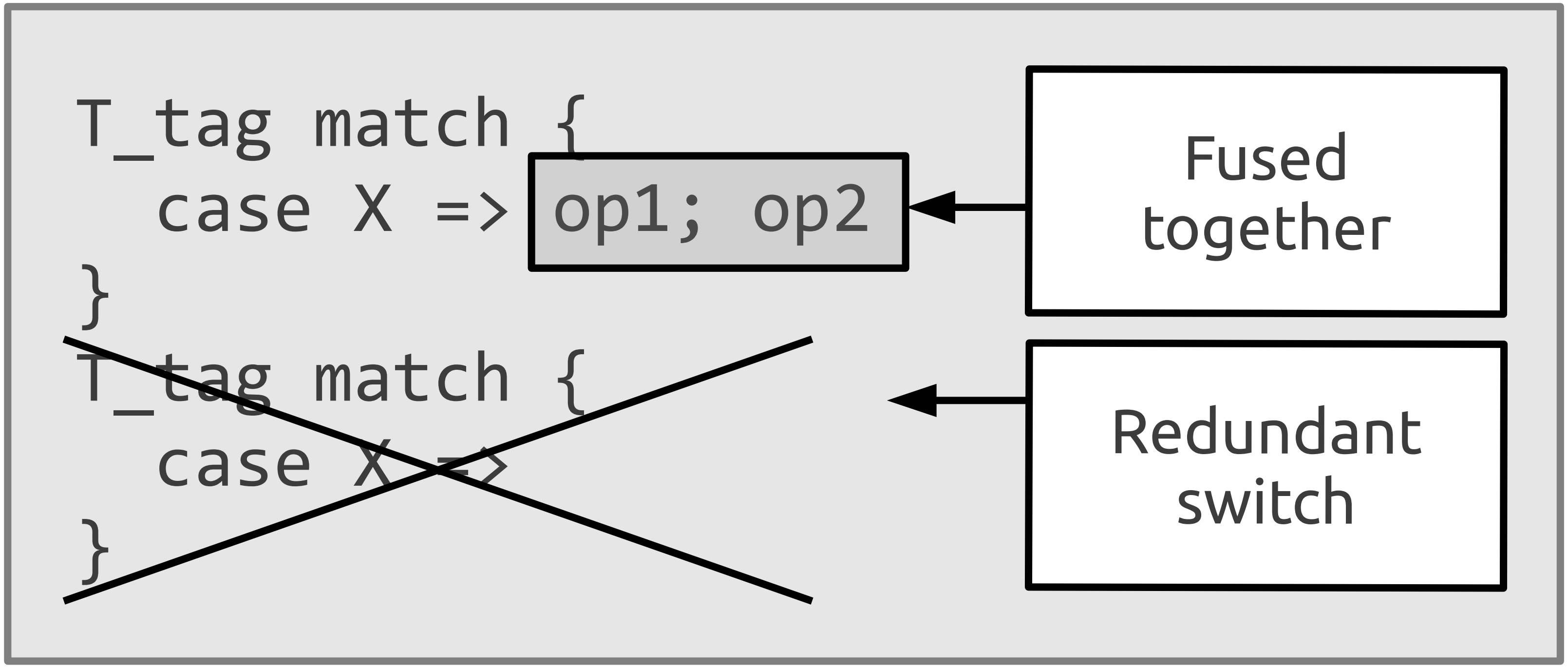
```
T_tag match {
  case X => op1; op2
}
```

~~T_tag match {~~
~~case X =>~~
~~}~~

Redundant switch

Switching on tags

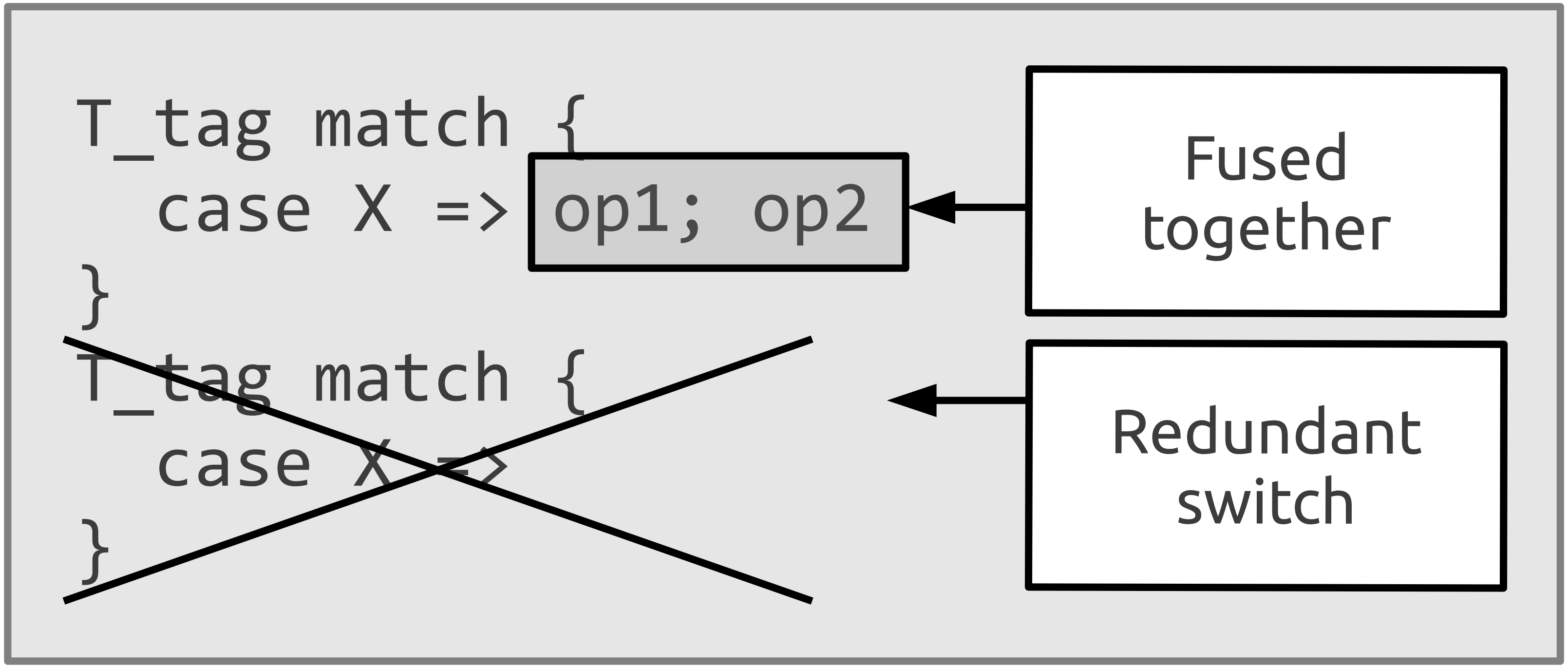
kills performance



```
T_tag match {
  case X => op1; op2
}
```

~~T_tag match {~~
~~case X =>~~
~~}~~

Switching on tags
kills performance



```
T_tag match {
  case X => op1; op2
}
```

~~T_tag match {~~
~~case X =>~~
~~}~~

This is critical for **array operations**

Switching

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
  }
}
```

Switching

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
  }
}
```

```
T_tag match {
  case INT => ...
  ...
}
```

Switching

`ArrayBuffer.reverse()`

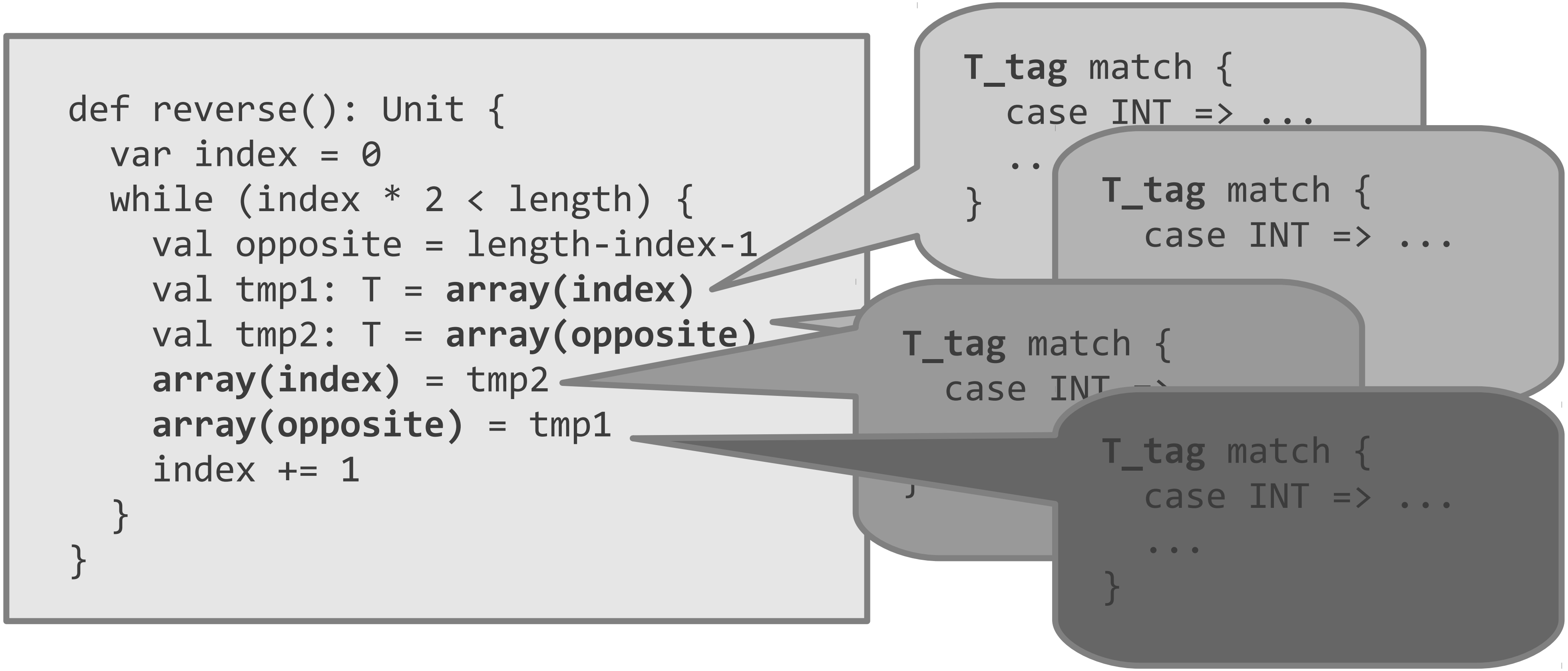
```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

```
T_tag match {
  case INT => ...
  ..
}
```

```
T_tag match {
  case INT => ...
  ...
}
```

Switching

```
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
  }
}
```

```
T_tag match {
  case INT => ...
  ..
}
```

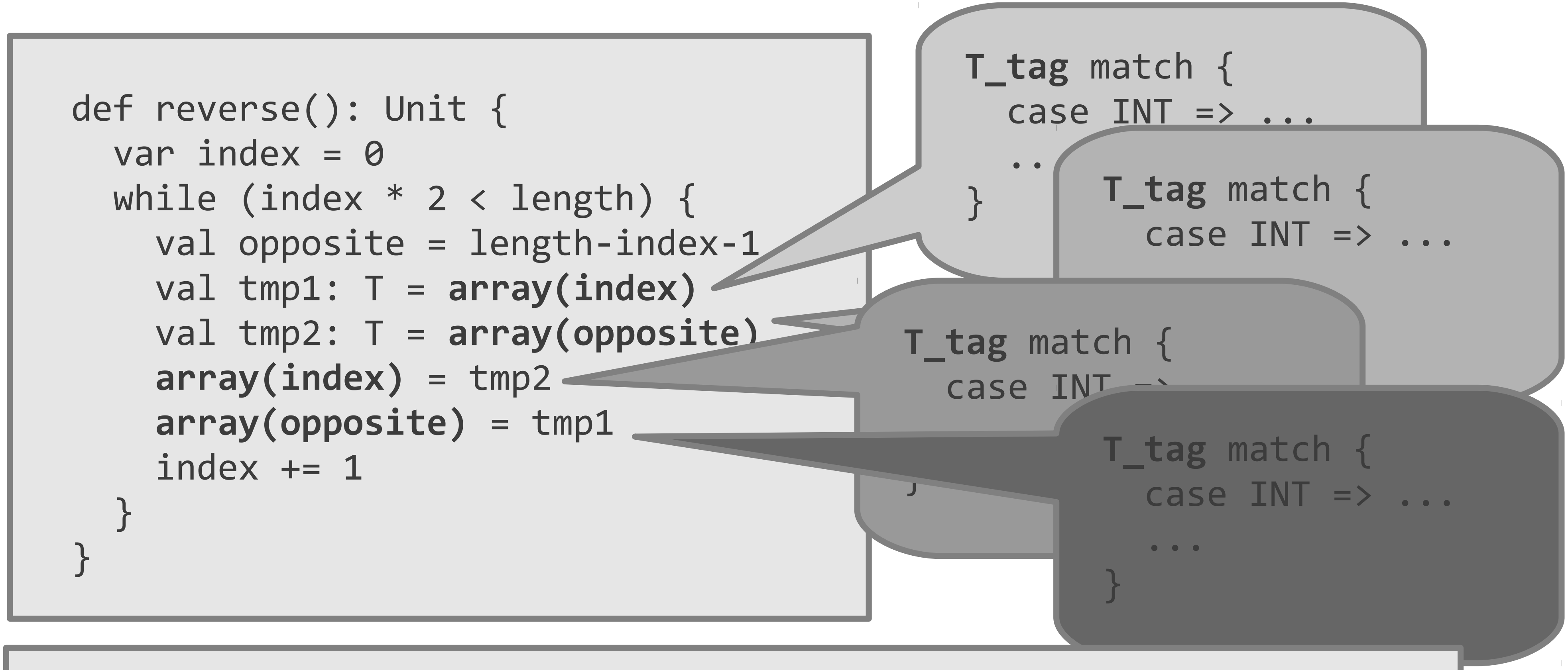
```
T_tag match {
  case INT => ...
```

```
T_tag match {
  case INT =>
```

```
T_tag match {
  case INT => ...
  ...
}
```

Switching

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
  }
}
```

```
T_tag match {
  case INT => ...
  ..
}
```

```
T_tag match {
  case INT => ...
```

```
T_tag match {
  case INT =>
}
```

```
T_tag match {
  case INT => ...
  ...
}
```

Fuse the operations together?

Switching

`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
  }
}
```

```
T_tag match {
  case INT =>
    val tmp1 = ...
    val tmp2 = ...
    array(.) = ...
    array(.) = ...
  ...
}
```

Switching

`ArrayBuffer.reverse()`

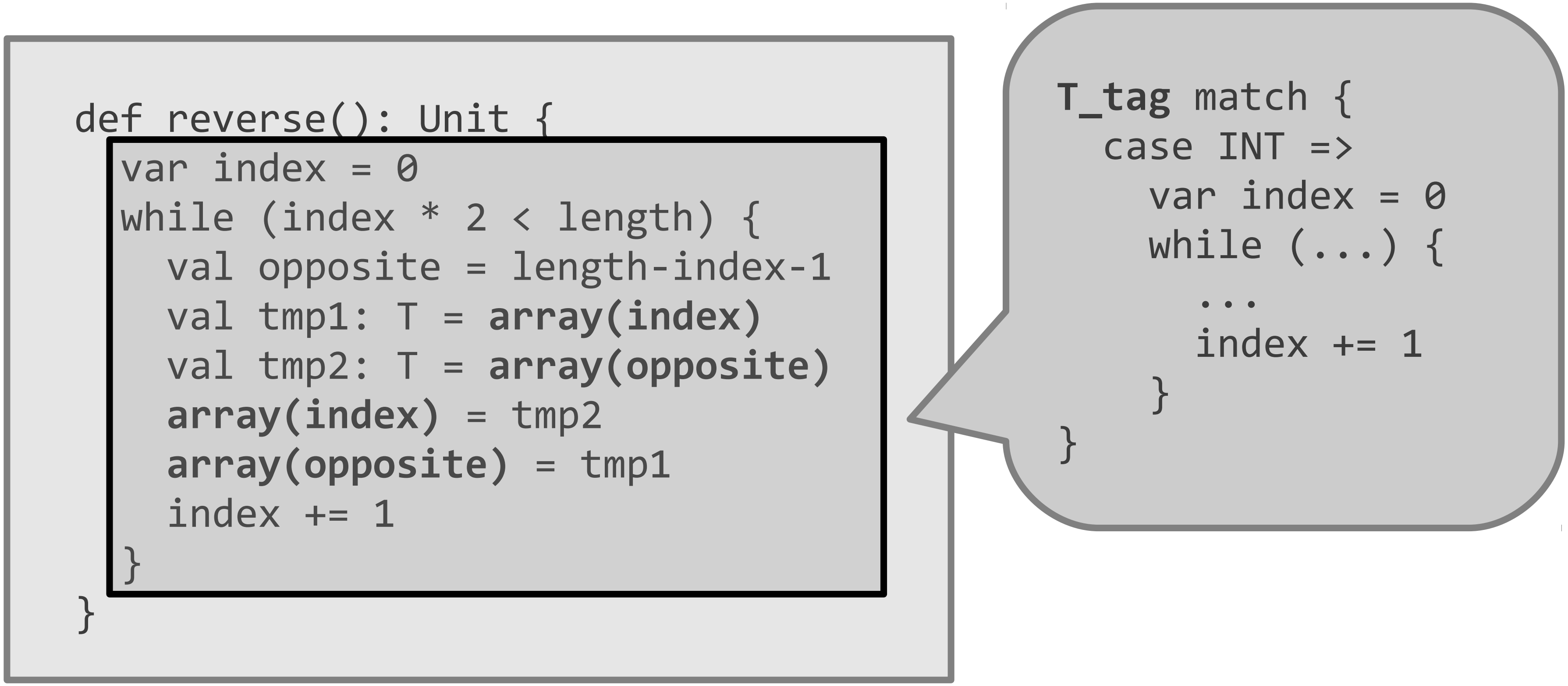
```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

```
T_tag match {
  case INT =>
    val tmp1 = ...
    val tmp2 = ...
    array(.) = ...
    array(.) = ...
  ...
}
```

Hoist the switch out of the loop?

Switching

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
  }
}
```

```
T_tag match {
  case INT =>
    var index = 0
    while (...) {
      ...
      index += 1
    }
}
```

Switching

ArrayBuffer.reverse()

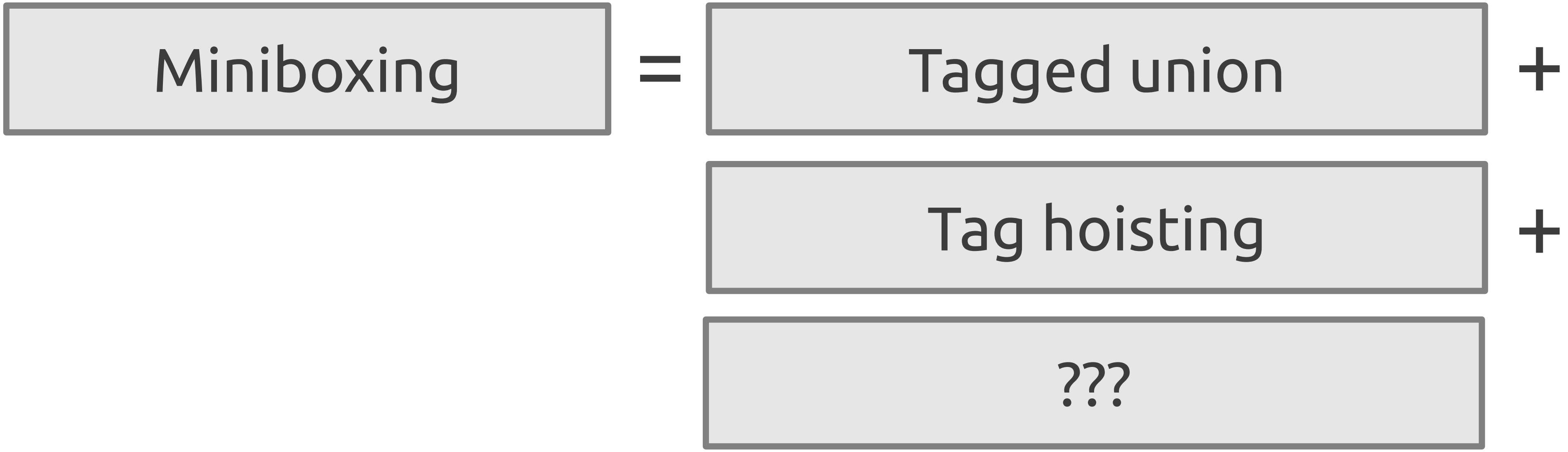
```
def reverse(): Unit {
  var index = 0
  while (index * 2 < length) {
    val opposite = length-index-1
    val tmp1: T = array(index)
    val tmp2: T = array(opposite)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

```
T_tag match {
  case INT =>
    var index = 0
    while (...) {
      ...
      index += 1
    }
}
```

Is that enough? Method may be **called from a loop**

Performance

needs one more ingredient



Miniboxing = Tagged union + Tag hoisting + ???

Performance

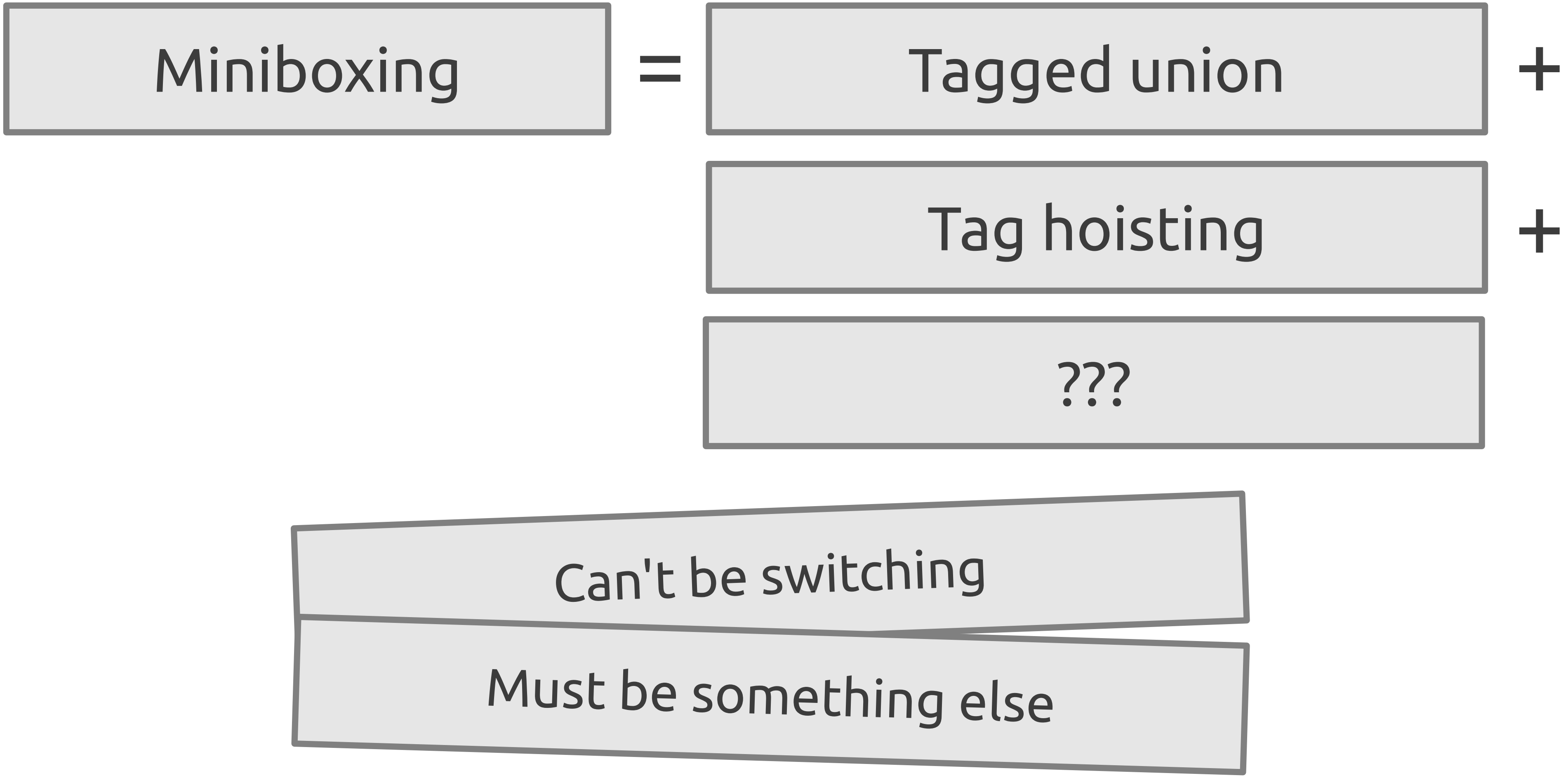
needs one more ingredient

Miniboxing = Tagged union + Tag hoisting + ???

Can't be switching

Performance

needs one more ingredient



Miniboxing = Tagged union + Tag hoisting + ???

Can't be switching

Must be something else

Dispatching

- Dispatch object
  - Encodes array interactions

Dispatching

- Dispatch object
  - Encodes array interactions

```
class Dispatcher[T] {
  def array_get(...): Long
  def array_set(...): Unit
}
```

Dispatching

- Dispatch object
  - Encodes array interactions

```
class Dispatcher[T] {
  def array_get(...): Long
  def array_set(...): Unit
}
```

```
def identity_M(T_dispatcher: Dispatcher[T],
               t: Long): Long
```

Dispatching

- Dispatch object
  - Encodes array interactions

```
class Dispatcher[T] {
  def array_get(...): Long
  def array_set(...): Unit
}
```

```
def identity_M(T_dispatcher: Dispatcher[T],
                t: Long): Long
```

instead of tag

Dispatching

- Dispatch object
  - Encodes array interactions

```
object IntDispatcher extends Dispatcher[Int] {
  def array_get(...): Long = ...
  def array_set(...): Unit = ...
}
```

Dispatching

- Dispatch object
  - Encodes array interactions

```
object IntDispatcher extends Dispatcher[Int] {
  def array_get(...): Long = ...
  def array_set(...): Unit = ...
}
```

```
object LongDispatcher ...
object CharDispatcher ...
```

Dispatching

- Dispatch object
  - Encodes array interactions

```
object IntDispatcher extends Dispatcher[Int] {
  def array_get(...): Long = ...
  def array_set(...): Unit = ...
}
```

```
object LongDispatcher ...
object CharDispatcher ...
```

Passing a dispatcher = hoisted already

Dispatching

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(other)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

Dispatching

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(other)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

T_dispatcher.array_get

Dispatching

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(other)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

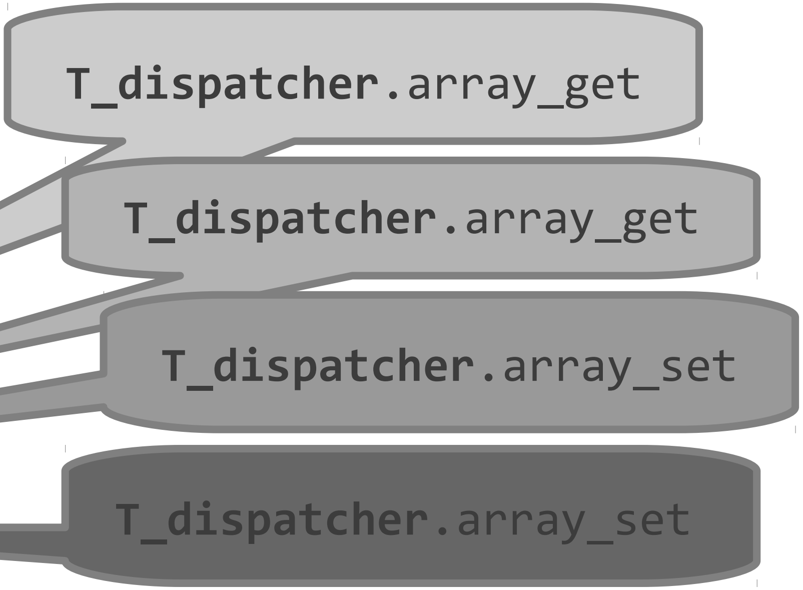
T_dispatcher.array_get

T_dispatcher.array_get

Dispatching

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(other)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```



T_dispatcher.array_get

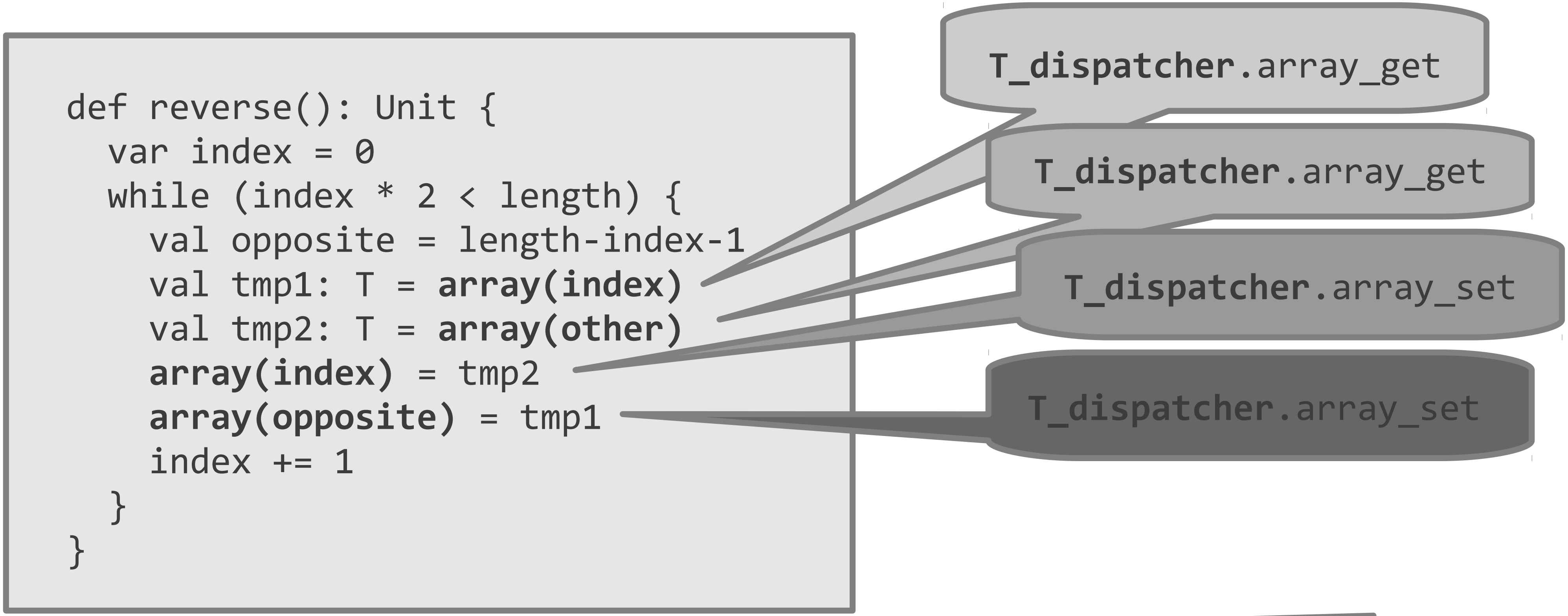
T_dispatcher.array_get

T_dispatcher.array_set

T_dispatcher.array_set

Dispatching

`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(other)
    array(index) = tmp2
    array(opposite) = tmp1
    index += 1
  }
}
```

With inlining, we get good performance

Dispatching

```
ArrayBuffer.reverse()
```

```
T_dispatcher.array_get
```

Dispatching

```
ArrayBuffer.reverse()
```

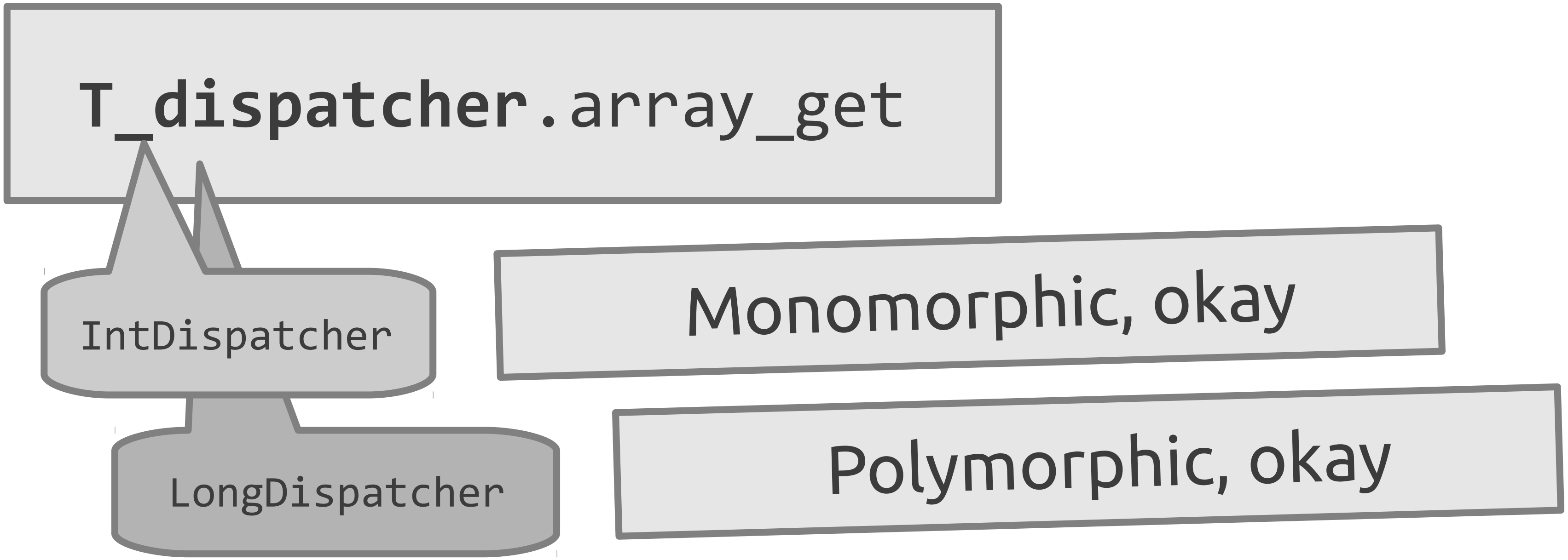
```
T_dispatcher.array_get
```

```
IntDispatcher
```

Monomorphic, okay

Dispatching

```
ArrayBuffer.reverse()
```



T_dispatcher.array_get

IntDispatcher

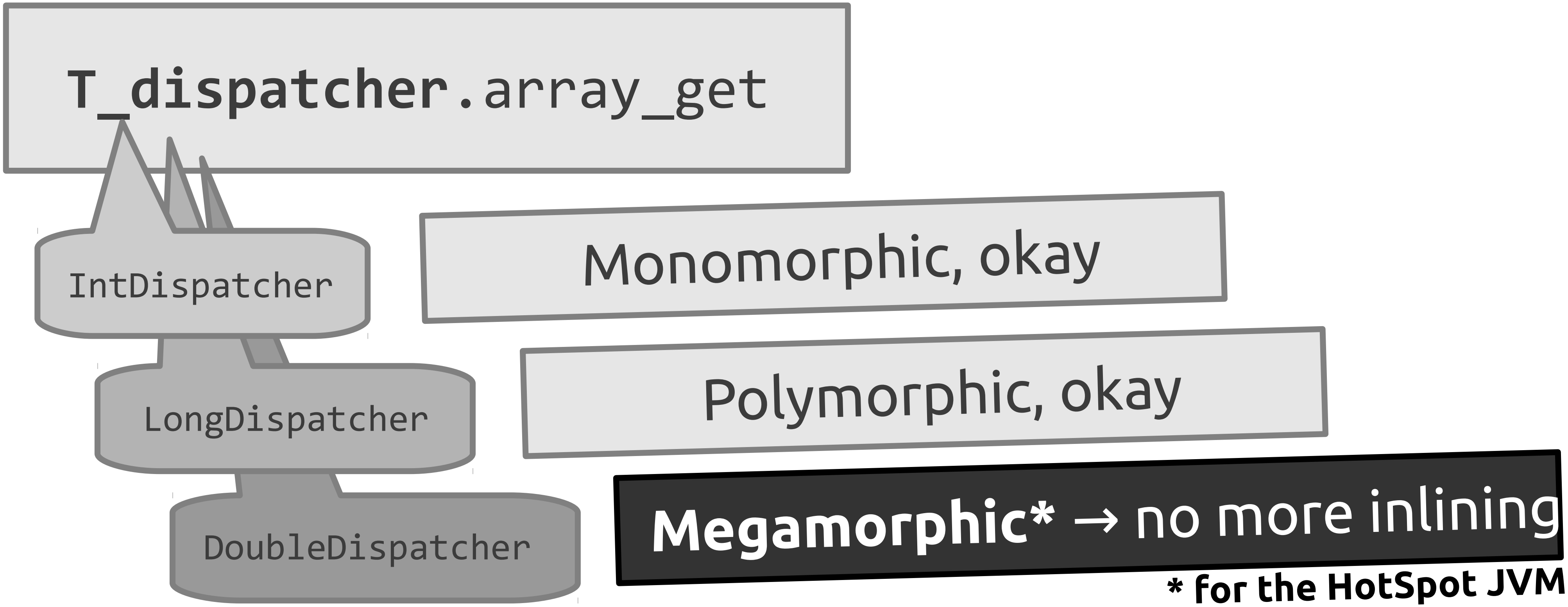
LongDispatcher

Monomorphic, okay

Polymorphic, okay

Dispatching

`ArrayBuffer.reverse()`



T_dispatcher.array_get

IntDispatcher

LongDispatcher

DoubleDispatcher

Monomorphic, okay

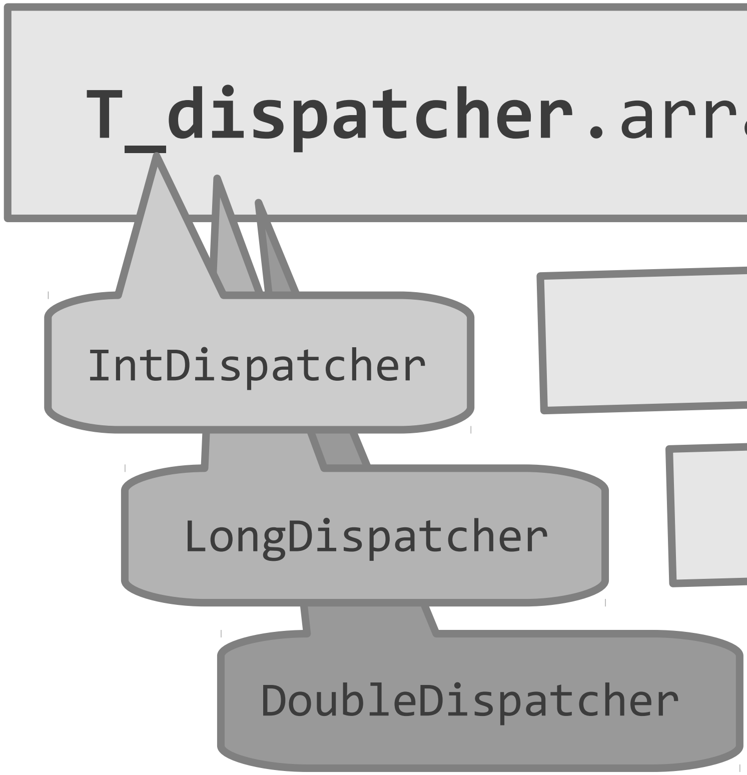
Polymorphic, okay

Megamorphic* → no more inlining

*** for the HotSpot JVM**

Dispatching

`ArrayBuffer.reverse()`



`T_dispatcher.array_get`

`IntDispatcher`

`LongDispatcher`

`DoubleDispatcher`

Monomorphic, okay

Polymorphic, okay

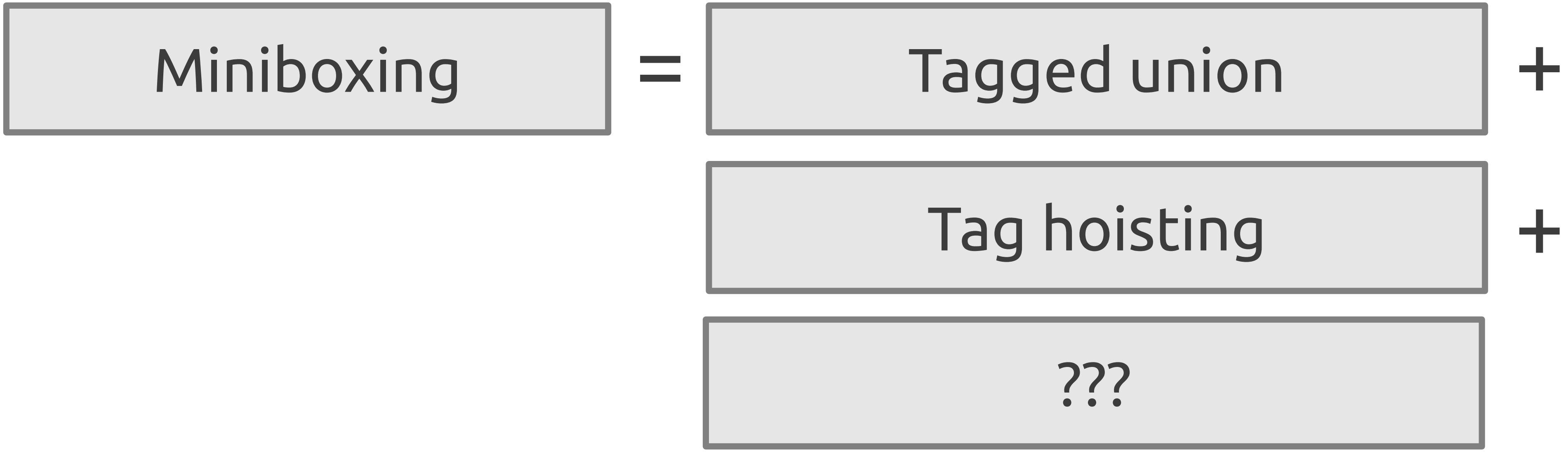
Megamorphic* → no more inlining

*** for the HotSpot JVM**

No more inlining → bad performance

Performance

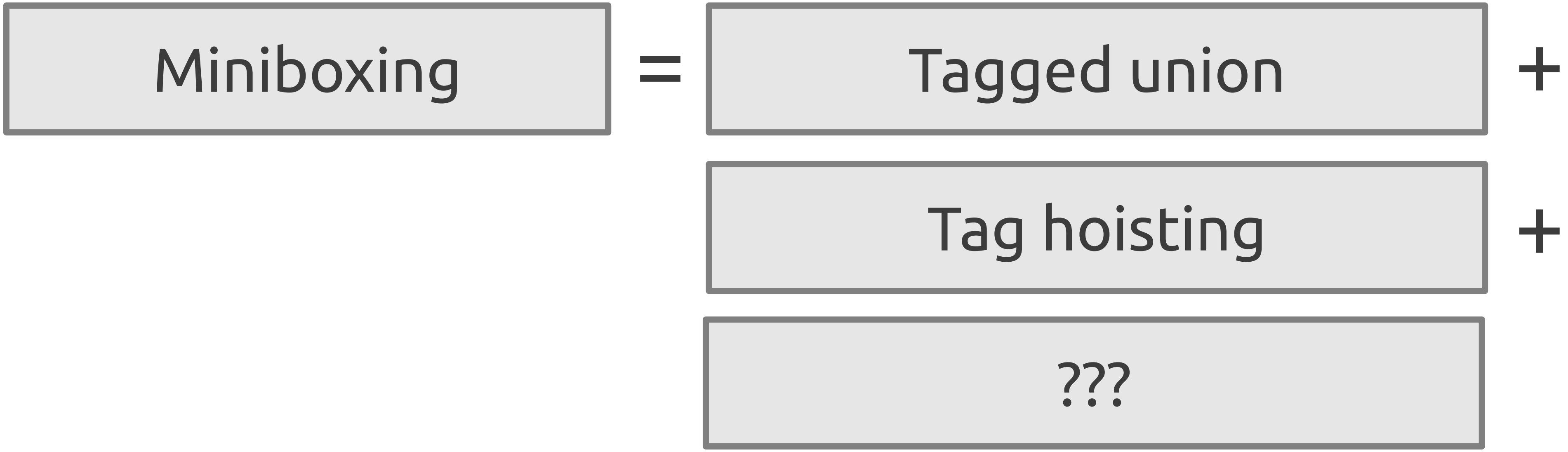
needs one more ingredient



Miniboxing = Tagged union + Tag hoisting + ???

Performance

needs one more ingredient



Miniboxing = Tagged union + Tag hoisting + ???

Object oriented dispatch isn't that

The secret ingredient

The secret ingredient

- **Switch-based** dispatching

```
T_tag match {
  case INT => ...
  ...
}
```

The secret ingredient

- **Switch-based** dispatching
- When instantiating the class
  - **T_tag** is **known**

```
T_tag match {
  case INT => ...
  ...
}
```

The secret ingredient

- **Switch-based** dispatching
- When instantiating the class
  - **T_tag** is **known**
  - **T_tag** is a **constant**

```
T_tag match {
  case INT => ...
  ...
}
```

The secret ingredient

- **Switch-based** dispatching
- When instantiating the class
  - **T_tag** is **known**
  - **T_tag** is a **constant**

```
T_tag match {
  case INT => ...
  ...
}
```

Encode T_tag in the class name?

The secret ingredient

- **Switch-based** dispatching
- When instantiating the class
  - **T_tag** is **known**
  - **T_tag** is a **constant**

```
T_tag match {
  case INT => ...
  ...
}
```

Encode T_tag in the class name?

Staticly? Code explosion!

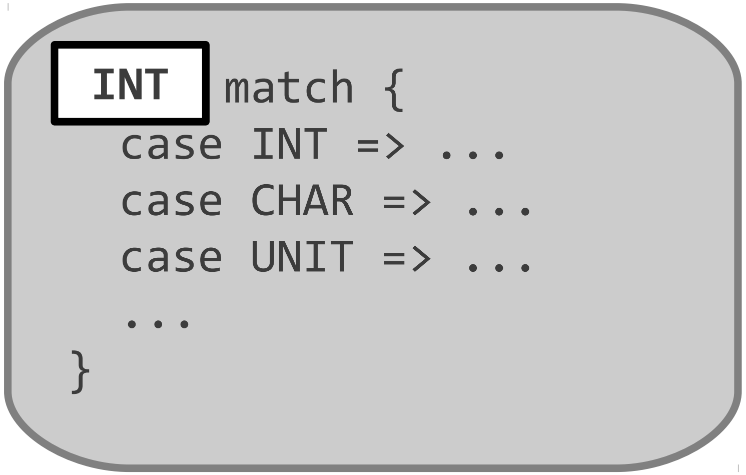
Load-time specialization

- Load-time transformation

```
T_tag match {
  case INT => ...
  case CHAR => ...
  case UNIT => ...
  ...
}
```

Load-time specialization

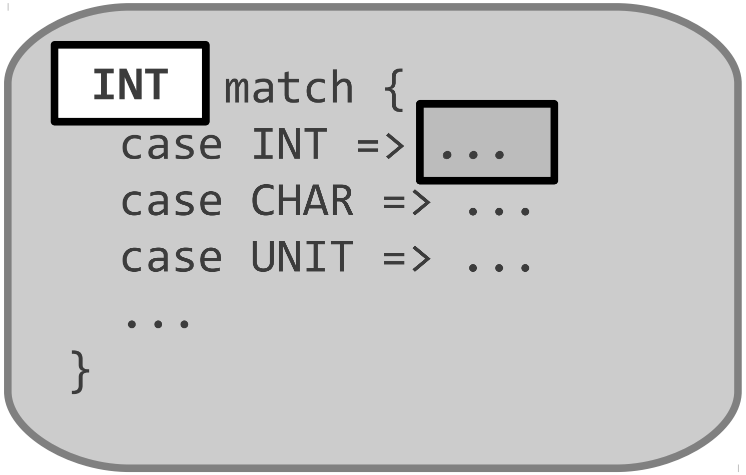
- Load-time transformation
  - set **T_tag** statically



```
INT match {
  case INT => ...
  case CHAR => ...
  case UNIT => ...
  ...
}
```

Load-time specialization

- Load-time transformation
  - set **`T_tag`** statically
  - perform **constant folding**



```
INT  match {
  case INT =>  ...
  case CHAR =>  ...
  case UNIT =>  ...
  ...
}
```

Load-time specialization

- Load-time transformation
  - set **`T_tag`** statically
  - perform **constant folding**
  - perform **dead code elimination**

```
...
```

Load-time specialization

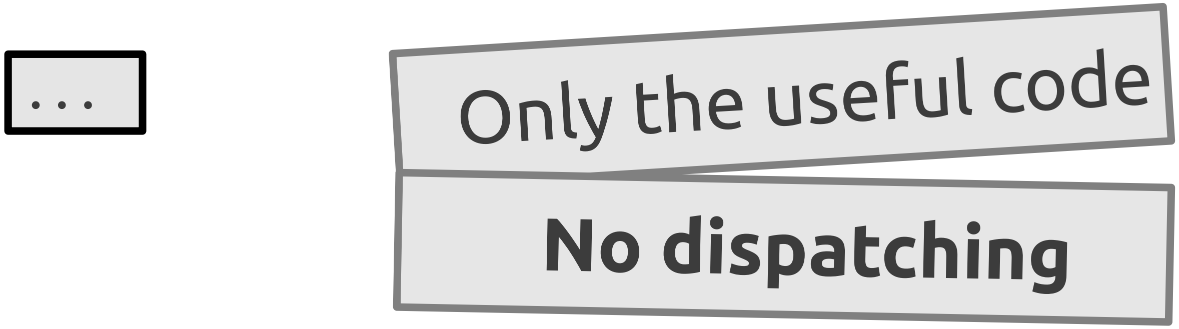
- Load-time transformation
  - set `T_tag` statically
  - perform **constant folding**
  - perform **dead code elimination**

...

Only the useful code

Load-time specialization

- Load-time transformation
  - set **`T_tag`** statically
  - perform **constant folding**
  - perform **dead code elimination**



...

Only the useful code

No dispatching

Load-time specialization

- Load-time transformation
  - set `T_tag` statically
  - perform **constant folding**
  - perform **dead code elimination**

...

Only the useful code

No dispatching

Is this the secret ingredient? **Yes!**

Performance

needs one more ingredient

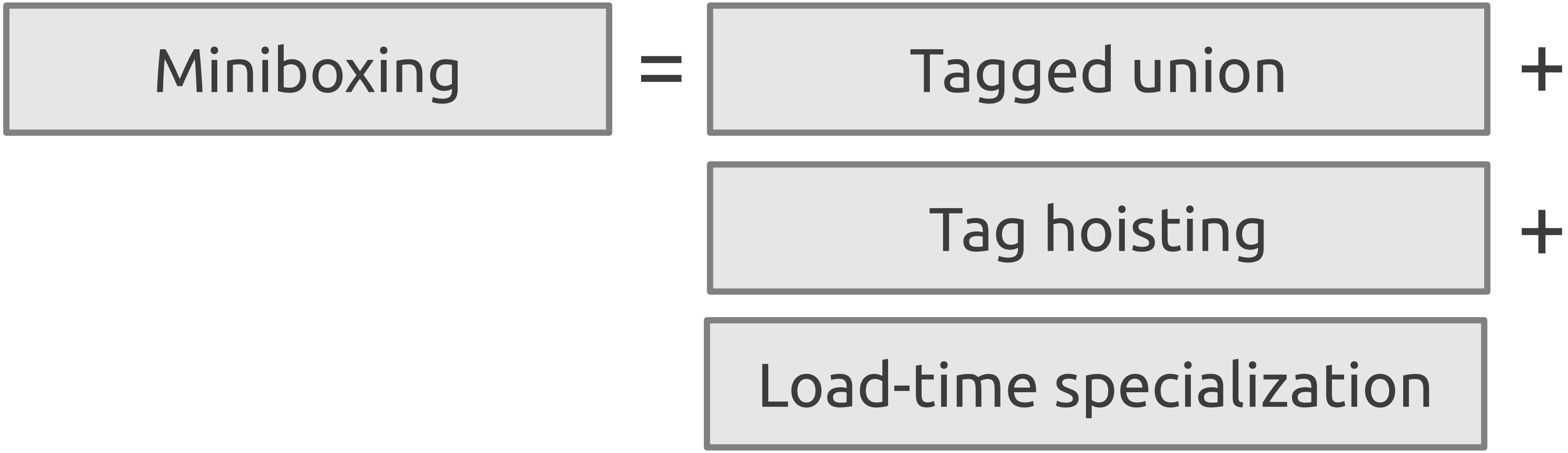
Miniboxing = Tagged union +

Tag hoisting +

Load-time specialization

Performance

needs one more ingredient



Miniboxing = Tagged union + Tag hoisting + Load-time specialization

Attaching tags to code enables load-time specialization

Generics

Specialization

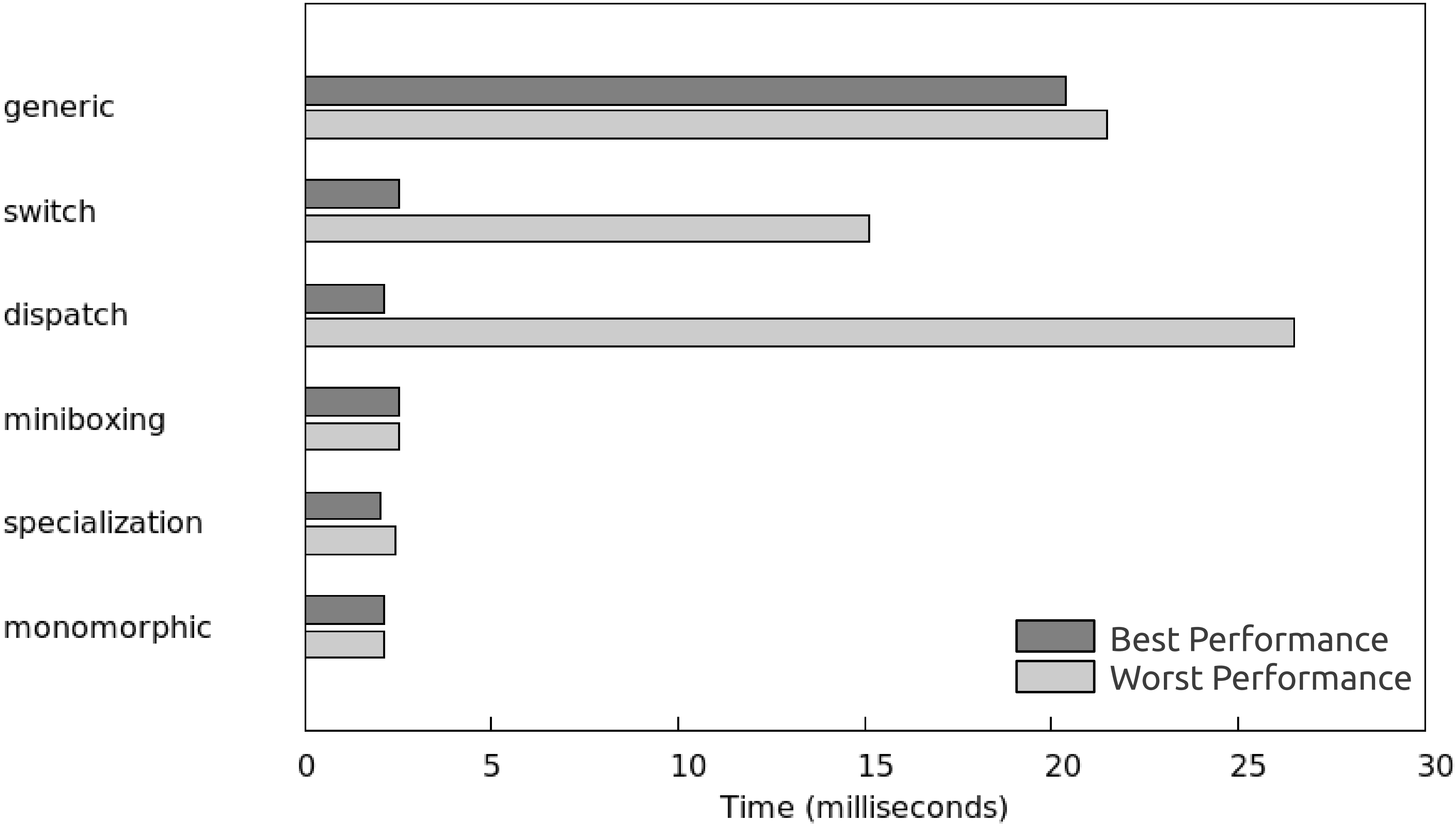
Miniboxing

Performance

Evaluation

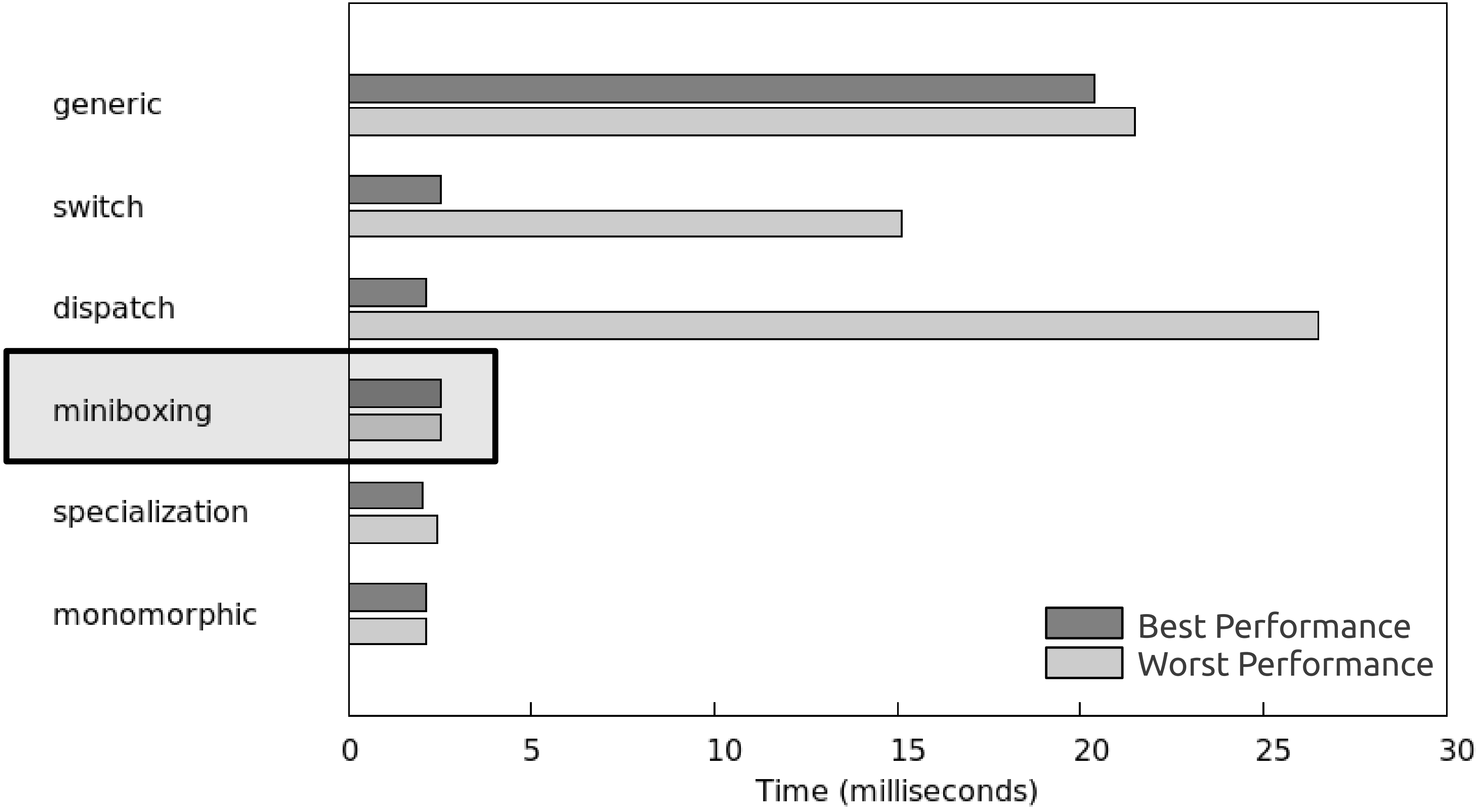
WE ARE HERE

Evaluation - Performance



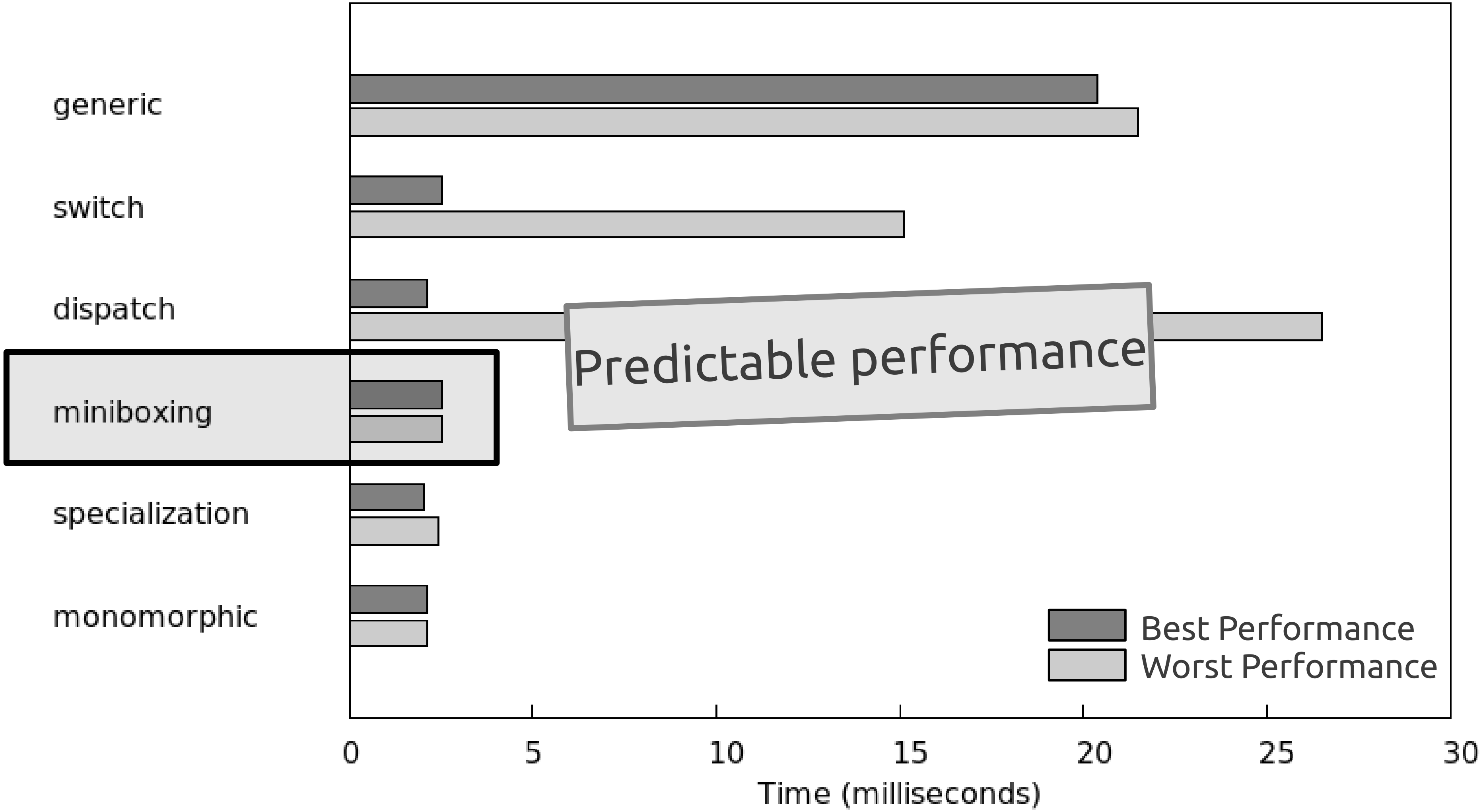
(less is better)

Evaluation - Performance



(less is better)

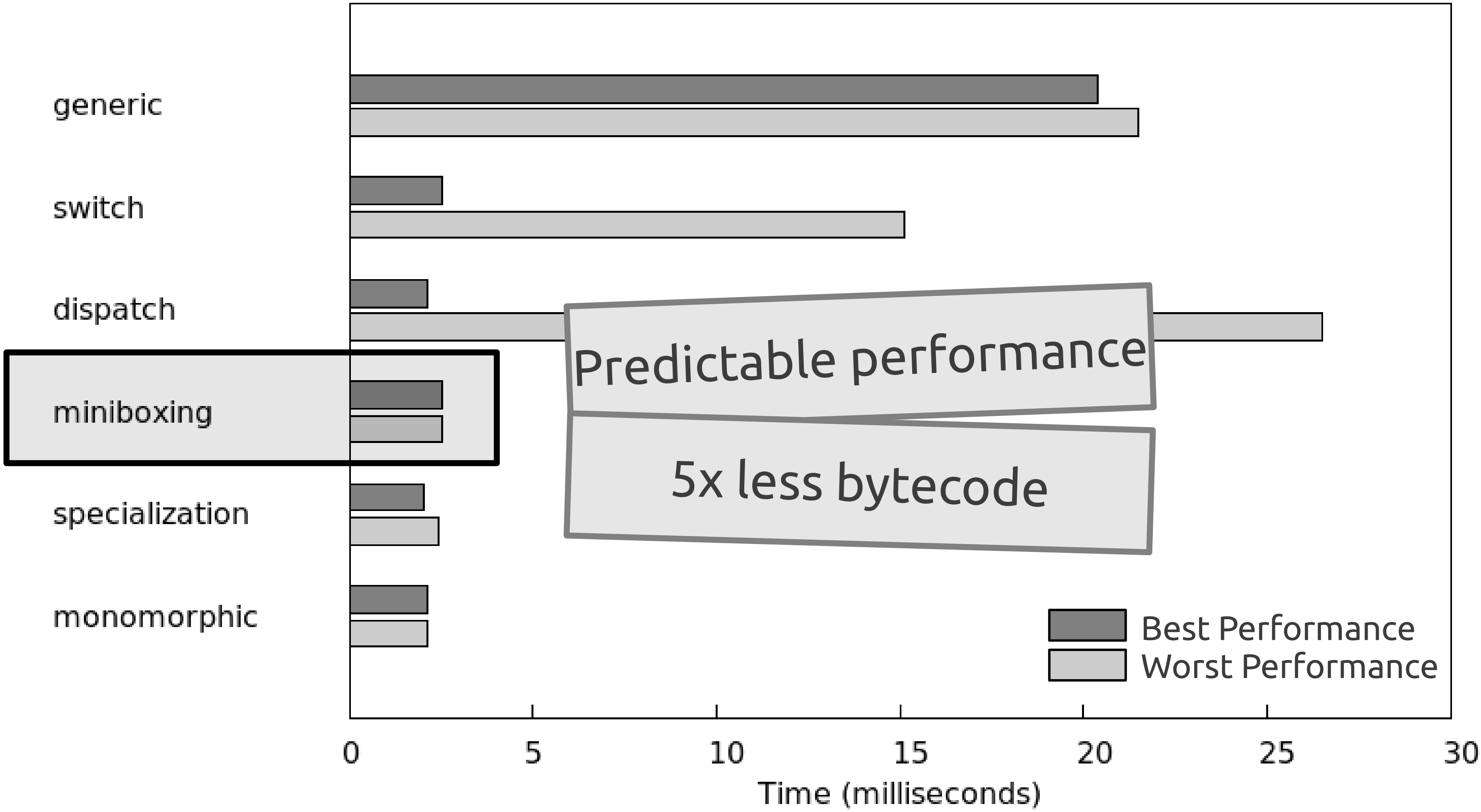
Evaluation - Performance



Predictable performance

(less is better)

Evaluation - Performance

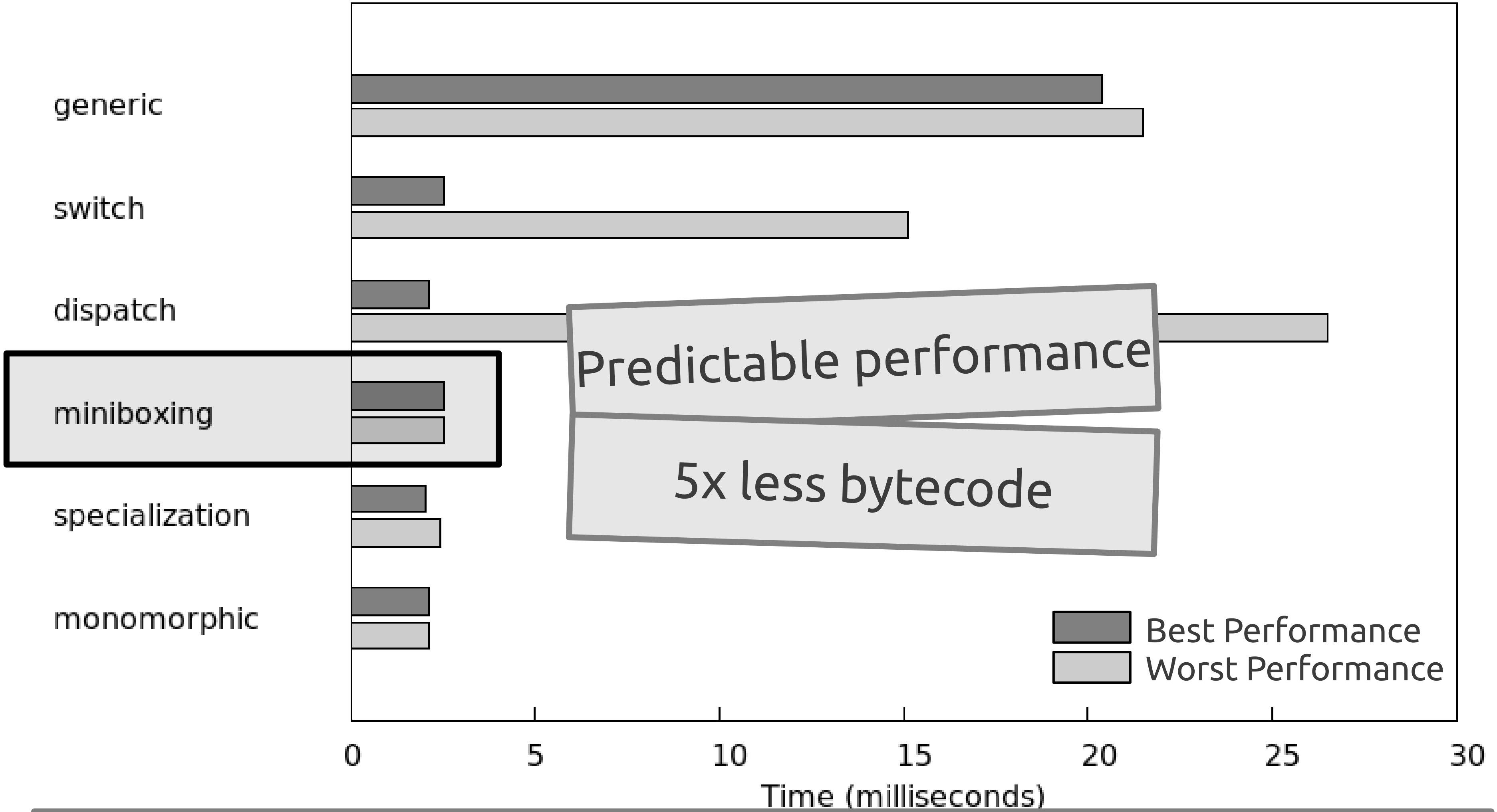


Predictable performance

5x less bytecode

(less is better)

Evaluation - Performance



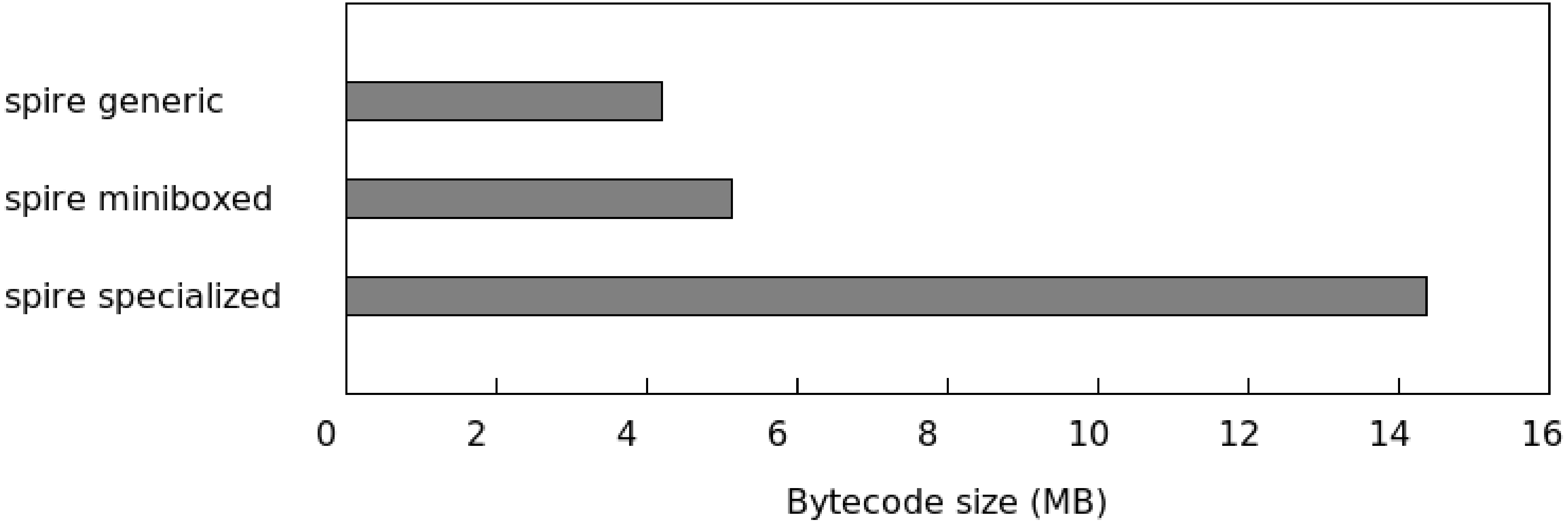
Predictable performance

5x less bytecode

Similar results on other benchmarks

Evaluation - Code size

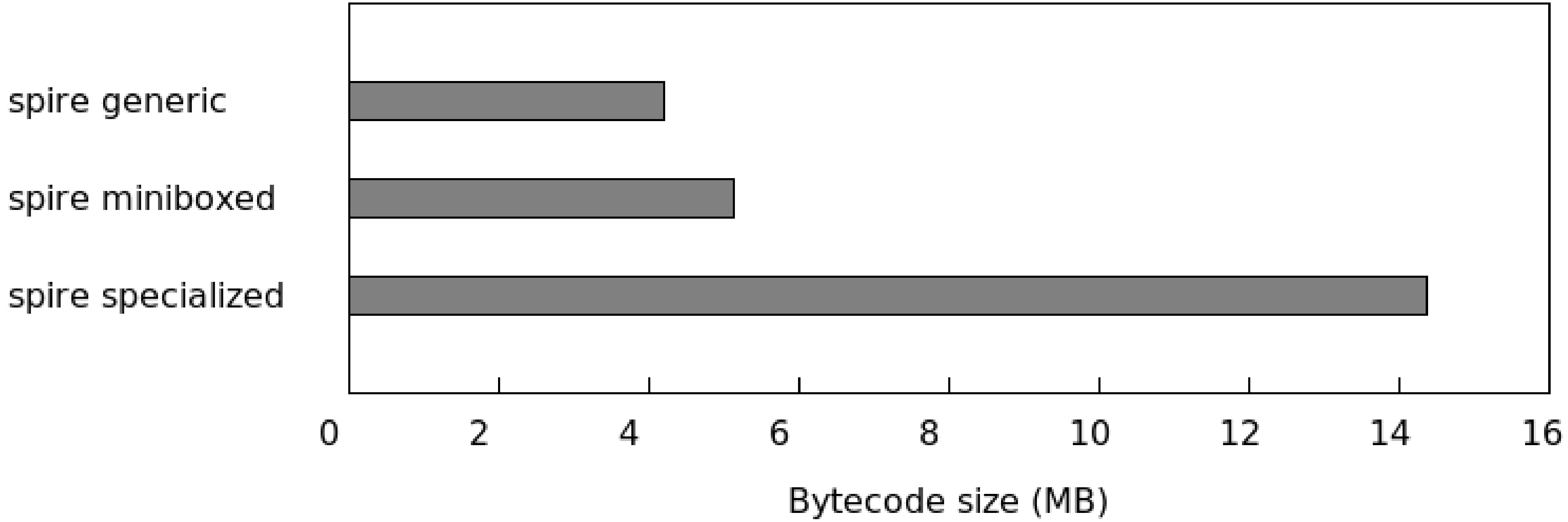
Spire – numeric abstractions library (12KLOC)



(less is better)

Evaluation - Code size

Spire – numeric abstractions library (12KLOC)



(less is better)

2.8x bytecode reduction (4.7x for `Vector` in std. lib)

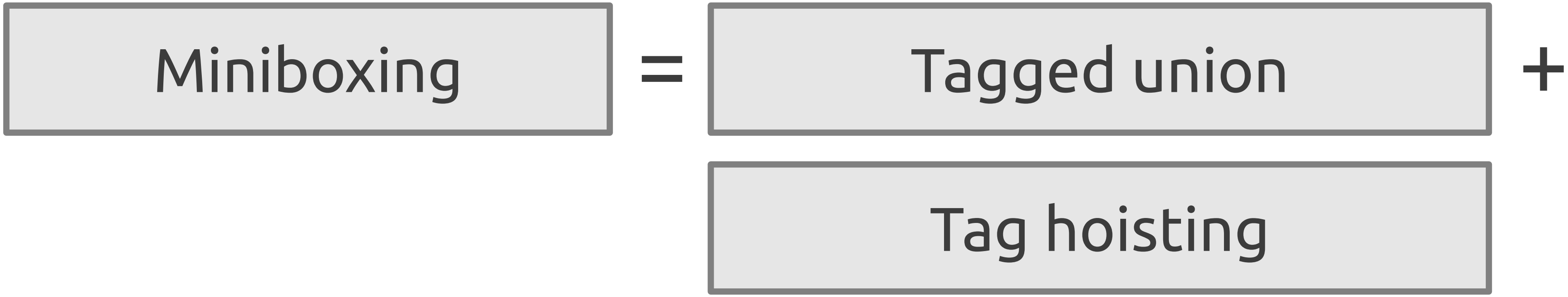
Contributions

Miniboxing

Contributions

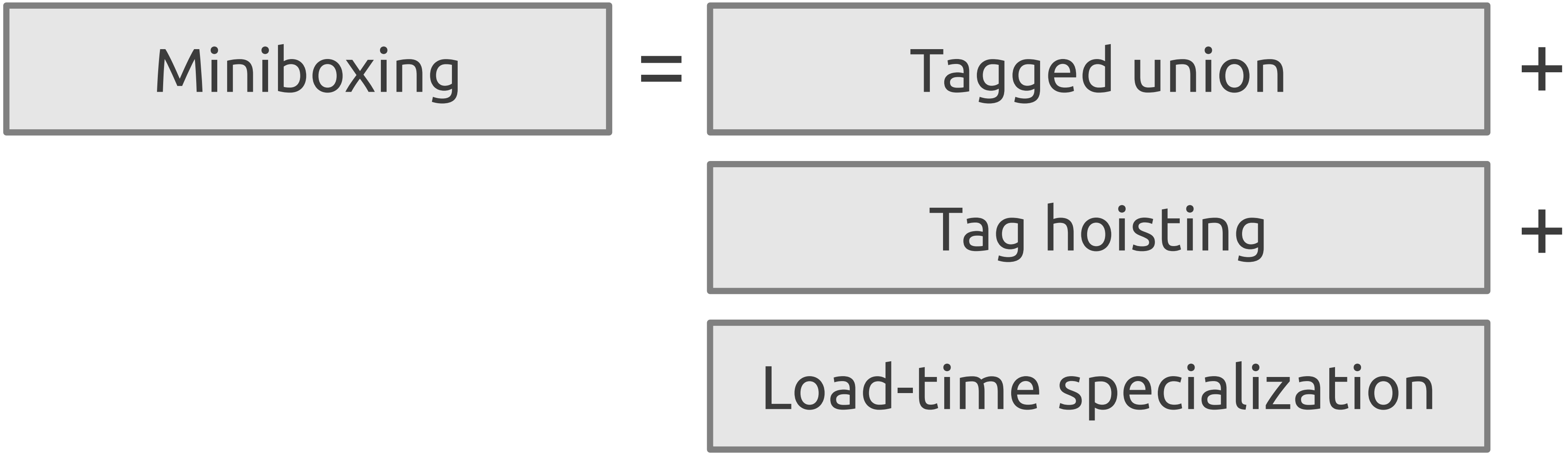
Miniboxing = Tagged union

Contributions



Miniboxing = Tagged union + Tag hoisting

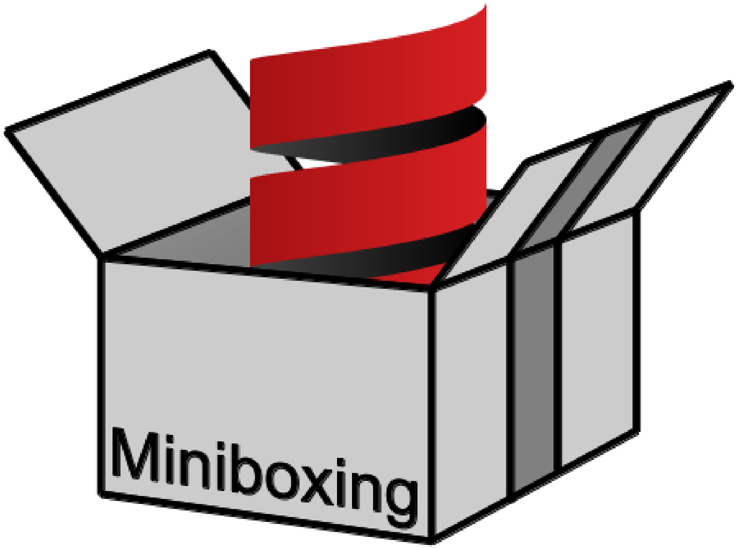
Contributions



Miniboxing = Tagged union + Tag hoisting + Load-time specialization

Conclusions

OOPSLA * Artifact * AEC * Evaluated *
Consistent * Complete * Well Documented * Easy to Reuse *



- improves performance
- reduces bytecode size

visit scala-miniboxing.org!