Project Valhalla:
The Good, the Bad and the Ugly
Vlad URECHE

PhD student in the Scala Team @ EPFL

Working on program transformations in the Scala programming language, focusing on **data representation**. Contributions to specialization, backend and Scaladoc.

@VladUreche
@VladUreche
vlad.ureche@epfl.ch
Vlad URECHE
PhD student in the Scala Team @ EPFL

Working on program transformations in the Scala programming language, focusing on **data representation**. Contributions to specialization, backend and Scaladoc.

**Miniboxing** is an alternative to the specialization transformation in scalac:
- Guides the programmer into maximizing performance
- Moderate increase in bytecode size
- Good performance

See [scala-miniboxing.org](http://scala-miniboxing.org).

@VladUreche
@VladUreche
vlad.ureche@epfl.ch
Vlad URECHE

PhD student in the Scala Team @ EPFL

Working on program transformations in the Scala programming language, focusing on data representation. Contributions to specialization, backend and Scaladoc.

Late Data Layout is the underlying technique used for:

- Programmer-driven transformations
- Miniboxing
- Multi-param value classes
- Compiler support for multi-stage programming

See scala-ldl.org.
Project Valhalla: The Good, the Bad and the Ugly
Project Valhalla

• Oracle project
  – started in 2014
  – goal: support for
    ● specialization and
    ● value classes
  – in the Java Virtual Machine
  – effort lead by Brian Goetz and John Rose

• No hard promises
  – but it might make it into JDK 10
Project Valhalla: The Good, the Bad and the Ugly
Project Valhalla: The Good, the Bad and the Ugly

- Generics Specialization
  - Very good performance
  - Predictable results
- Support for value classes
  - Even for multiple parameters
Project Valhalla: The Good, the Bad and the Ugly

- Incompatible with the Scala type system
- Variance
- Existential quantification
Project Valhalla:
The Good, the Bad and the Ugly

- We can support the Scala type system to some extent
- But some patterns are no longer compatible
Motivation
Generics
Compatibility
Limitations
Conclusion

Erasure
Valhalla
Miniboxing
Compilers
Compilers

Source code (language 1)

C, C++, Java, Scala, C#, Haskell, Clojure, ...
Compilers

Source code (language 1)
- C, C++, Java, Scala, C#, Haskell, Clojure, ...

Compiler

Source code (language 2)
- x86/ARM assembly
- LLVM bitcode
- Cuda code
- JVM bytecode
- .NET bytecode
- ...
Compilers

Source code (language 1)
C, C++, Java, Scala, C#, Haskell, Clojure, ...

Why so much fuss?

Source code (language 2)
x86/ARM assembly
LLVM bytecode
Cuda code
JVM bytecode
.NET bytecode
...
Compilers

Abstraction $\xrightarrow{\text{compiler}}$ Implementation
Compilers

Abstraction \[\rightarrow\text{compiler}\] Implementation

Variable
Compilers

Abstraction → compiler → Implementation

Variable → Stack slot or Processor register or Heap location
Compilers

Abstraction

Variable

compiler

Implementation

Stack slot or Processor register or Heap location

... Implementation details
Scala compiler

Scala Abstractions
Scala compiler

Scala Abstractions → scalac → JVM Bytecode
Scala compiler

Scala Abstractions → scalac → JVM Bytecode

Types
Scala compiler

Scala Abstractions → scalac → JVM Bytecode

Types → Classes Interfaces
Scala compiler

trait T1
trait T2
val t: T1 with T2 = new T1 with T2
trait T1
trait T2
val t: T1 with T2 = new T1 with T2

... class $anon extends T1 with T2 { ... }
val t: T1 = new $anon
trait T1
trait T2
val t: T1 with T2 = new T1 with T2

... class $anon extends T1 with T2 { ... }
val t: T1 = new $anon
trait T1
trait T2
val t: T1 with T2 = new T1 with T2

... class $anon extends T1 with T2 { ... }
val t: T1 = new $anon

instantiation class
Scala compiler

trait T1
trait T2
val t: T1 with T2 = new T1 with T2

... class $anon extends T1 with T2 {... }
val t: T1 = new $anon
For any `new` instantiation in the source code, there must exist a bytecode class that is instantiated.
For any **new** instantiation in the source code, there must exist a bytecode class that is instantiated.

For any **value** of a type in the source code, there must exist a class or interface that allows calling its methods.
Scala compiler

Scala Abstractions

scalac

JVM Bytecode

Types

Classes

Interfaces
Valhalla Proposal

- Presentation is based on an documented proposal
  - Submitted for review to the
    - scala-internals and
    - valhalla-dev mailing lists
- Proposal document: go.epfl.ch/valhalla
  - Not yet a pre-SIP, it's too early
Motivation
Generics
Compatibility
Limitations
Conclusion

Erasure
Valhalla
Miniboxing
Generic Implementation

new C[String]

new C[Int]

new C[Float]

new C[Double]

Generic Reference

Erasure
new C[String]
new C[Int]
new C[Float]
new C[Double]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
def foo[T](t: T) = new C[T](t)

def foo(t: Object): ... = new ...(t)
Generic Type Params

def foo[T](t: T) = new C[T](t)

def foo(t: Object): ... = new ...(t)

The decision must be made statically at compile-time.
def foo[T](t: T) = new C[T](t)

def foo(t: Object): C = new C(t)
Generic Implementation

new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

Generic Reference

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]

Erasure
Generic Implementation

new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

Generic Reference

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]

Erasure
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
class C[+T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)
class C[+T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)

class C(t: Object)
val c1: C = new C(???)
val c2: C = new C(???)
```scala
class C:+T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)

class C(t: Object)
val c1: C = new C(???)
val c2: C = new C(???)
```

Need to box values → overhead
Erasure > Performance

class C[+T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)

class C(t: Object)
val c1: C = new C(Integer.valueOf(3))
val c2: C = new C(Double.valueOf(3.0))
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
c: C[_]
This is the bytecode part.
new C : C
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Erasure > Abstraction

class C[+T](t: T)
val c1: C[_] = new C(3)
val c2: C[Any] = new C(3.0)
class $C[+T](t: T)$
val c1: $C[\_\_] = \text{new } C(3)$
val c2: $C[\text{Any}] = \text{new } C(3.0)$

class $C(t: \text{Object})$
val c1: $C = \text{new } C(\text{Integer.valueOf}(3))$
val c2: $C = \text{new } C(\text{Double.valueOf}(3.0))$
class C[+T](t: T)
val c1: C[[_]] = new C(3)
val c2: C[Any] = new C(3.0)

Both variance and existentials can be translated easily when using erasure.

class C(t: Object)
val c1: C = new C(Integer.valueOf(3))
val c2: C = new C(Double.valueOf(3.0))
def foo(t: Int => Int) = ...  
def identity[T]: T => T = ...  
foo(identity[Int])
def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])


def foo(t: Function1) = ...
def identity: Function1 = ...
foo(identity)
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- skipped in the presentation
- fully described in the proposal

[go.epfl.ch/valhalla]
# Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Volhalla</th>
<th>Miniboiling</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is it always possible to instantiate the type desired. e.g. not for Array[T], which needs a ClassTag in scope for the instantiation.
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Minikube</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is it always possible to instantiate the type desired. e.g. not for Array[T], which needs a ClassTag in scope for the instantiation.
### Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Is it always possible to instantiate the type desired. e.g. not for Array[T], which needs a ClassTag in scope for the instantiation?**

**Depending on the approach to implementing generics, classes can be optimized for:**
- primitive types
- value classes
- both
<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Volhalla</th>
<th>Minikicking</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is it always possible to instantiate the type desired. e.g. not for Array[T], which needs a ClassTag in scope for the instantiation.

Depending on the approach to implementing generics, classes can be optimized for:
- primitive types
- value classes
- both
# Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Minikubing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is it always possible to instantiate the type desired, e.g. not for Array[T], which needs a ClassTag in scope for the instantiation?

Depending on the approach to implementing generics, classes can be optimized for:
- primitive types
- value classes
- both

Can code behave differently based on the type arguments used to instantiate it?
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxying</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>worst</td>
<td>best</td>
<td>worst</td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>best</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>best</td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>best</td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>worst</td>
<td>worst</td>
<td>worst</td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>worst</td>
<td>worst</td>
<td>worst</td>
</tr>
</tbody>
</table>

Is it always possible to instantiate the type desired. e.g. not for Array[T], which needs a ClassTag in scope for the instantiation.

Depending on the approach to implementing generics, classes can be optimized for:
- primitive types
- value classes
- both

Can code behave differently based on the type arguments used to instantiate it?
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can we fix this? Enter Project Valhalla.
Generic Implementation

new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

Generic Reference

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
c: C[___]

Erasure
Erasure

Generic Implementation

new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C

Forces boxed arguments.

Generic Reference

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
c: C[___]
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]
c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
c: C[___]
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C_{T=int}
new C_{T=float}

new C

: C

Valhalla
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C_{T=int}
new C_{T=double}
new C_{T=Float}

new C : C
new C_{T=int} : C_{T=int}
new C_{T=double} : C_{T=double}

c : C[T]
c : C[String]
c : C[Int]
c : C[Float]
c : C[Double]
c : C[___]
new C[T]

new C[String]

new C[Int]

new C[Float]

new C[Double]

new C_{T=double}

new C_{T=float}

new C_{T=int}

c: C[T]

c: C[String]

c: C[Int]

c: C[Float]

c: C[Double]

c: C[__]
Java forbids the equivalent of C[T] and C[\_]
Valhalla>Performance

class C[any +T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)
class C[any +T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)
class C\[any \ +T\](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)
class C[any +T](t: T)
val c1: C[Int] = new C(3)
val c2: C[Double] = new C(3.0)

val c1: C = new C_{T=int}(3)
val c2: C = new C_{T=double}(3.0)
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
new C_{T=double} : C_{T=double}
new C_{T=float}   : C_{T=float}
new C_{T=int}    : C_{T=int}

bytecode for the erased version + metadata necessary to instantiate the class for value types

instantiated at loading-time from the bytecode template
<table>
<thead>
<tr>
<th>Properties</th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
class C[any +T](t: T)
val c: C[ ] =
  if (…)
    new C[Int](3)
  else
    new C[Double](3.0)
class C[any +T](t: T)
val c: C[_] = if (...) new C[Int](3) else new C[Double](3.0) // C_{T=int}
class C[any +T](t: T)
val c: C[_] = 
if (...) 
    new C[Int](3) // C_{T=int}
else 
    new C[Double](3.0) // C_{T=double}
new C[T]
new C[String]
new C[Int]
new C<Float]
new C[Double]

new C_{T=int}
new C_{T=float}
new C_{T=double}

new C: C

???
c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]

???
c: C[___]
Different classes, nothing in common among them.
Valhalla>Abstraction

class C[any +T](t: T)
val c1: C[_] = 
  if (...) 
    new C[Int](3) // C_{T=int}
  else 
    new C[Double](3.0) // C_{T=double}

Valhalla\textgreater Abstraction

class C\[any \ +T\](t: T)
val c1: C\[\_\] = 
  if (…)
    new C[Int](3)
  else
    new C[Double](3.0)

// Object :(
// C\{_T=int\}
// C\{_T=double\}
Valhalla>Abstraction

def foo(t: Int => Int) = ...
def identity[T]: T=> T = ...
foo(identity[Int])
def foo(t: Int => Int) = ...  
def identity[T]: T=> T = ...  
foo(identity[Int])

def foo(t: Function1_{T1=int,R=int}) = ...  
def identity: Function1 = ...  
foo(identity[Int])
Valhalla>Abstraction

```scala
def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])
```

```scala
def foo(t: Function1_{T1=int,R=int}) = ...
def identity: Function1 = ...
foo(identity[Int])
```

Mismatch, the erased `Function1` and the instantiated template `Function1_{T1=...}` are incompatible.
Valhalla>Abstraction

• What does it mean to you?
  – We lost definition-site variance (class C[T])
  – We lost existential quantification over values (C[\_])
  – Specialization is all-or-nothing
    • .NET experience → not all code is worth specializing
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*No go for Scala*
Miniboxing addresses that
Generic Implementation

- `new C[T]`
- `new C[String]`
- `new C[Int]`
- `new C[Float]`
- `new C[Double]`

new `C_L`

- `new C_J`
- `new C_D`

Generic Reference

- `c: C[T]`
- `c: C[String]`
- `c: C[Int]`
- `c: C[Float]`
- `c: C[Double]`
- `c: C[___]`
Generic Implementation

new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C_L : C
new C_J : C
new C_D : C

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]

c: C[__]
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C_L : C
new C_J : C
new C_D : C

Common interface shared by all miniboxed classes

c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]
c: C[_]
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Miniboxing > Performance

- Good baseline performance
Miniboxing>Performance

- Good baseline performance
- But there are slowdowns
  - Object version used for primitive types
For T = Int

- new C[T]
- new C[String]
- new C[Int]
- new C[Float]
- new C[Double]

new C_L : C
new C_J : C
new C_D : C

- c: C[T]
- c: C[String]
- c: C[Int]
- c: C[Float]
- c: C[Double]
- c: C[___]
For $T = \text{Int}$

- new $C[T]$
- new $C[\text{String}]$
- new $C[\text{Int}]$
- new $C[\text{Float}]$
- new $C[\text{Double}]$

$c : C[T]$

$c : C[\text{String}]$

$c : C[\text{Int}]$

$c : C[\text{Float}]$

$c : C[\text{Double}]$

$c : C[\_\_]$
new C[T]  
new C[String]  
new C[Int]  
new C[Float]  
new C[Double]

For T = Int

new C_L : C  
new C_J : C  
new C_D : C

c: C[T]  
c: C[String]  
c: C[Int]  
c: C[Float]  
c: C[Double]

c: C[___]
For $T = \text{Int}$

new C[T]

c: C[T]

The object version of C acts as type C[Int] → suboptimal
Miniboxing>Performance

- Good baseline performance
- But there are slowdowns
  - Object version used for primitive types
Miniboxing>Performance

• Good baseline performance
• But there are slowdowns
  – **Object version used for primitive types**
  – Valhalla ➔ impossible, but we lose all abstraction
Miniboxing Performance

- Good baseline performance
- But there are slowdowns
  - Object version used for primitive types
  - Valhalla → impossible, but we lose all abstraction
  - Specialization →
Miniboxing>Performance

```scala
scala> class C[@specialized T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> new C(3)
res0: C[Int] = C$mcI$sp@33b1594e // specialized version

scala> newC(3)
res1: C[Int] = C@7de0a5c6 // generic version
```
Miniboxing > Performance

- Good baseline performance
- But there are slowdowns
  - **Object version used for primitive types**
  - Valhalla → impossible, but we lose all abstraction
  - Specialization → slowdown, no warning
Miniboxing > Performance

- Good baseline performance
- But there are slowdowns
  - **Object version used for primitive types**
  - Valhalla → impossible, but we lose all abstraction
  - Specialization → slowdown, no warning
  - Miniboxing →
scala> class C[@miniboxed T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
<console>:8: warning: The following code could benefit from miniboxing specialization if the type parameter T of method newC would be marked as "@miniboxed T" (it would be used to instantiate miniboxed type parameter T of class C)
  def newC[T](t: T) = new C(t)

  newC: [T](t: T)C[T]

scala> new C(3)
res0: C[Int] = C_J@5b649d6f // specialized version

scala> newC(3)
<console>:10: warning: The method newC would benefit from miniboxing type parameter T, since it is instantiated by a primitive type.
  newC(3)

res1: C[Int] = C_L@38013f5a // generic version
What it's trying to say:
Hey, you're about to shoot yourself in the foot! Watch out!

```
scala> def newC[T](t: T) = new C(t)
<console>:8: warning: The following code could benefit from miniboxing specialization if the type parameter T of method newC would be marked as "@miniboxed T" (it would be used to instantiate miniboxed type parameter T of class C)
   def newC[T](t: T) = new C(t)
   ^

newC: [T](t: T)C[T]

scala> new C(3)
res0: C[Int] = C_J@5b649d6f // specialized version

scala> newC(3)
<console>:10: warning: The method newC would benefit from miniboxing type parameter T, since it is instantiated by a primitive type.
   newC(3)
   ^

res1: C[Int] = C_L@38013f5a // generic version
```
Miniboxing>Performance

What it's trying to say:
Hey, you're about to shoot yourself in the foot! Watch out!

```
scala> def newC[T](t: T) = new C(t)
<console>:8: warning: The following code could benefit from miniboxing specialization if the type parameter T of method newC would be marked as "@miniboxed T" (it would be used to instantiate miniboxed type parameter T of class C)
  def newC[T](t: T) = new C(t)
  ^
newC: [T](t: T)C[T]
```

What it's trying to say:
Should I call a doctor?

```
scala> newC(3)
<console>:10: warning: The method newC would benefit from miniboxing type parameter T, since it is instantiated by a primitive type.
  newC(3)
  ^
res1: C[Int] = C_J@5b649d6f // specialized version
```

```
scala> new C(3)
res0: C[Int] = C_J@5b649d6f // specialized version
```
Miniboxing > Performance

- Good baseline performance
- But there are slowdowns
  - **Object version used for primitive types**
  - Valhalla → impossible, but we lose all abstraction
  - Specialization → slowdown, no warning
  - Miniboxing → warns, but still slows down

Heed the warnings!
Miniboxing > Performance

- Examples
  - “Clash of the Lambdas” Scala benchmarks (vs Java8)
    - improved by 2.5-14x over generic (arxiv.org)
  - “Optimistic Re-Specialization” (Tom Switzer)
    - improved by 10x over generic (io.pellucid.com)
  - Many other benchmarks
    - on the website scala-miniboxing.org
# Properties

<table>
<thead>
<tr>
<th>Properties</th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generic Implementation

- new C[T]
- new C[String]
- new C[Int]
- new C[Float]
- new C[Double]

new C_L : C

Generic Reference

- c: C[T]
- c: C[String]
- c: C[Int]
- c: C[Float]
- c: C[Double]
- c: C[___]
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td>good</td>
<td></td>
<td>worst</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Miniboxing > Abstraction

class C[@miniboxed T](t: T)
val c1: C[_] =
  if (...)
    new C[Int](3)  // bytecode: C_J
  else
    new C[Double](3.0)  // bytecode: C_D
Miniboxing > Abstraction

class C[@miniboxed T](t: T)
val c1: C[_] = if (...) new C[Int](3) else new C[Double](3.0) // bytecode: C // bytecode: C_J // bytecode: C_D
Miniboxing > Abstraction

class C[@miniboxed T](t: T)
val c1: C[_] =
  if (…)
    new C[Int](3)
  else
    new C[Double](3.0)
  // bytecode: C
  // bytecode: C_J
  // bytecode: C_D

Miniboxing
Miniboxing > Abstraction

def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])
Miniboxing > Abstraction

def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])

def foo(t: Function1) = ...
def identity: Function1 = ...
foo(identity)
Miniboxing > Abstraction

def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])

def foo(t: Function1) = ...
def identity: Function1 = ...
foo(identity)
### Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td></td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td></td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>best</td>
<td><strong>worst</strong></td>
<td>good</td>
<td></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>

Either solution has a drawback. What about combining them?
Motivation

Generics

Compatibility

Limitations

Conclusion

Erasure

Valhalla

Miniboxing
Generic Implementation

- `new C[T]`
- `new C[String]`
- `new C[Int]`
- `new C[Float]`
- `new C[Double]`

Generic Reference

- `c: C[T]`
- `c: C[String]`
- `c: C[Int]`
- `c: C[Float]`
- `c: C[Double]`

Valhalla
Generic Implementation

new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

Generic Reference

???

???
c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]

Specialized

Generic

Ragnarök
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C_{T=Int}
new C_{T=Float}
new C_{T=Double}

new C

: C

: C_{T=Int}
: C_{T=Float}
: C_{T=Double}

: C

Ragnarök

Generic Implementation

Generic Reference

???
???
c: C[T]
c: C[String]
c: C[Int]
c: C[Float]
c: C[Double]

c: C[___]
• **Instantiated templates** are specialized
  – But contain a compatibility layer
• **Generic interface** abstracts over the compat. layer
  – At the cost of losing optimality
  – Some limitations still apply
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>worst</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>
# Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>best</td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>worst</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>
Ragnarök\textgreater Performance

- Good baseline performance
- But there are slowdowns
  - Object version used for primitive types
  - Valhalla $\rightarrow$ impossible
  - Specialization $\rightarrow$ no warning, slowdown
  - Miniboxing $\rightarrow$ warns, but still slows down
Generic Implementation

- new C[T]
- new C[String]
- new C[Int]
- new C[Float]
- new C[Double]

Generic Reference

- C_interface
- c: C[T]
- c: C[String]
- c: C[Int]
- c: C[Float]
- c: C[Double]

Specialized

- new C_{T=int}
- new C_{T=float}
- new C_{T=double}

Generic

- new C

Ragnarök
Ragnarök Performance

- Good baseline performance
- But there are slowdowns
  - **Object version used for primitive types**
  - Valhalla → impossible, but we lose all abstraction
  - Specialization → slowdown, no warning
  - Miniboxing → warns, but still slows down
  - Ragnarok → **like miniboxing**
# Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td><strong>best</strong></td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>worst</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>
# Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>worst</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
</tbody>
</table>
new C_{T=double}
new C_{T=float}
new C_{T=int}
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>good</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>
class C[T](t: T)
val c1: C[(_: _)] = 
  if (...) 
    new C[Int](3)  // C_{\{T=int\}}
  else 
    new C[Double](3.0)  // C_{\{T=double\}}
class C[T](t: T)
val c1: C[[_]] =
  if (...) 
    new C[Int](3)
  else 
    new C[Double](3.0)
  // C_{T=int}
  // C_{T=double}
// C_interface
// C_{T=int}
// C_{T=double}
Ragnarök -> Abstraction

class C[T](t: T) {
    val c1: C[_] = 
    if (...) 
        new C[Int](3) 
    else 
        new C[Double](3.0)

    // C_interface
    // C_{T=int}
    // C_{T=double}
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>good</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td></td>
</tr>
</tbody>
</table>
# Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>good</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
</tbody>
</table>
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Minibo\n\xing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Code size</strong></td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>good</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td><strong>goodish</strong></td>
</tr>
<tr>
<td><strong>Instantiation</strong></td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Applicability</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td><strong>Type-based decisions</strong></td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
</tbody>
</table>
Motivation

Generics

Compatibility

Limitations

Conclusion

Erasure
Valhalla
Miniboxing
Ragnarök

Limitations

```scala
scala> class C[any T](t: T)
defined class C
```
Ragnarök> Limitations

```scala
scala> class C[any T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]
```
Ragnarök> Limitations

```scala
scala> class C[any T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> val c1: C[Int] = new C(3)
c1: C[Int] = C_{T=int}@33b1594e
```
Ragnarök> Limitations

```
scala> class C[any T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> val c1: C[Int] = new C(3)
c1: C[Int] = C_{T=int}@33b1594e

scala> val c2: C[Int] = newC(3)
```
Ragnarök>Limitations

```scala
scala> class C[any T](t: T) defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> val c1: C[Int] = new C(3)
c1: C[Int] = C_{T=int}@33b1594e

scala> val c2: C[Int] = newC(3)
<console>:10: error: Type mismatch:
  found:    C[Int] (generic)
  expected: C[Int] (specialized)
    val c1: C[Int] = new C(3)
    ^
```
new C[T]
new C[String]
new C[Int]
new C[Float]
new C[Double]

new C_{T=int}
new C_{T=Float}
new C_{T=Double}

C_interface

Specialized
Generic

Ragnarök
Generic Implementation

- new C[T]
- new C[String]
- new C[Int]
- new C[Float]
- new C[Double]

Generic Reference

- C_interface
- c: C[T]
- c: C[String]
- c: C[Int]
- c: C[Float]
- c: C[Double]

Ragnarök

Specialized

Generic

Instantiate in generic context with T = Int
new C[T]

Generic Implementation

Instantiate in generic context with T = Int

new C

C_interface

: C

: C_{T=int}

Generic Reference

c: C[T]

c: C[Int]
Generic Implementation

new C[T]

new C

: C

: C_f{T=Int}

Generic Reference

C_interface

c: C[T]

c: C[Int]

Ragnarök

Specialized

Generic
Generic Implementation

new C[T]

new C

Instantiate in generic context with T = Int

C_interface

c: C[T]

C_{T=int}

c: C[Int]

C_{T=int} is a stronger promise than C_interface

Generic Reference

Ragnarökk

Specialized

Generic
**Generic Implementation**

- Instantiate in generic context with $T = \text{Int}$

- `new C[T]`

- `new C`

- `C\_interface`

- `c: C[T]`

- `c: C_{T=\text{int}}`

- `C\_interface` is not a subtype of `C_{T=\text{int}}`

**Generic Reference**

- `C_{T=int}` is a subtype of `C\_interface`

- `C\_interface` is a stronger promise than `C\_interface`

- `c: C[\text{Int}]`

- `C\_interface` is not a subtype of `C_{T=\text{int}}`
scala> class C[any T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> val c1: C[Int] = new C(3)
c1: C[Int] = C_{T=int}@33b1594e

scala> val c2: C[Int] = newC(3)
<console>:10: error: Type mismatch:
  found:    C[Int] (generic)
  expected: C[Int] (specialized)
  val c1: C[Int] = new C(3)
  ^
Limitations

scala> class C[any T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> val c1: C[Int] = new C(3)
c1: C[Int] = C_{T=int}@33b1594e

scala> val c2: C[Int] = newC(3)
<console>:10: error: Type mismatch:
found:    C[Int] (generic)     // C_interface
expected: C[Int] (specialized)  // C_{T=int}

val c1: C[Int] = new C(3)
  ^
Ragnarök>Limitations

scala> class C[any T](t: T)
defined class C

scala> def newC[T](t: T) = new C(t)
newC: [T](t: T)C[T]

scala> val c1: C[Int] = new C(3)
c1: C[Int] = C_{T=int}@33b1594e

scala> val c2: C[Int] = newC(3)
<console>:10: error: Type mismatch:
  found:    C[Int] (generic)               // C_interface
  expected: C[Int] (specialized)         // C_{T=int}
val c1: C[Int] = new C(3)

At bytecode level:
C_{T_int} is a subtype of C_interface
C_interface is not a subtype of C_{T=int}
Motivation

Generics

Compatibility

Limitations

Conclusion

Erasure
Valhalla
Miniboxing
Conclusion

- Ragnarök Proposal: go.epfl.ch/valhalla
- Valhalla
  - Load-time instantiation support
- Miniboxing
  - Compatibility scheme
  - Warnings
- Still in progress
  - Abstractions (addressing the limitations)
Thank you!
## Properties

<table>
<thead>
<tr>
<th></th>
<th>Erasure</th>
<th>Valhalla</th>
<th>Miniboxing</th>
<th>Ragnarök</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Code size</td>
<td>best</td>
<td>good</td>
<td>worst</td>
<td>good</td>
</tr>
<tr>
<td>Abstraction</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td><strong>goodish</strong></td>
</tr>
<tr>
<td>Instantiation</td>
<td>best</td>
<td>worst</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Applicability</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Type-based decisions</td>
<td>worst</td>
<td>best</td>
<td>good</td>
<td>good</td>
</tr>
</tbody>
</table>
Ragnarök>Limitations

def foo(t: Int => Int) = ...
def identity[T]: T=> T = ...
foo(identity[Int])
Ragnarök>Limitations

def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])

def foo(t: Function1_{T1=int,R=int}) = ...
def identity: Function1_interface = ...
foo(identity)
Ragnarök: Limitations

def foo(t: Int => Int) = ...
def identity[T]: T => T = ...
foo(identity[Int])

def foo(t: Function1_{T1=int,R=int}) = ...
def identity: Function1_interface = ...
foo(identity)

Mismatch:
  expected: Function1_{T1=int,R=int}
  found: Function1_interface
scala> def foo(f: Int => Int) = f(3)
foo: (t: Int => Int)Int

scala> def identity[T]: T=>T = (x: T) => x
identity: [T]=> T => T

scala> foo(identity[Int])
<console>:10: error: The result of method identity is a generic Function1[T, T], which is instantiated for T=Int. However, this is not compatible with the specialized Function1[Int, Int]. A fix would be to mark the type parameter T of method identity as “any T”:
    foo(identity[Int])
        ^
def foo(t: Int => Int) = ...
def identity[any T]: T => T = ...
foo(identity[Int])
Ragnarök>Limitations

```scala
def foo(t: Int => Int) = ...
def identity[any T]: T => T = ...
foo(identity[Int])
```
Ragnarök

Limitations

```scala
def foo(t: Any => Nothing) = ...
// works with and without "any":
def identity[T]: T => T = ...
foo(identity[Int])
```
def foo(t: Any => Nothing) = ...

// works with and without "any":
def identity[T]: T=> T = ...
foo(identity[Int])
Ragnarök > Limitations

```scala
def foo(f: Int => Int) = f(3)
foo: (t: Int => Int)Int

def identity[T]: T=>T = (x: T) => x
identity: [T]=> T => T

foo(identity[Int])
<console>:10: error: The result of method identity is a generic Function1[T, T], which is instantiated for T=Int. However, this is not compatible with the specialized Function1[Int, Int]. A fix would be to mark the type parameter T of method identity as “any T”:

foo(identity[Int])
```