Late Data Layout:

Unifying Data Representation Transformations

11th of September 2014
Virtual Machine Meetup
ETH Zürich, Switzerland

scala-miniboxing.org/ldl

Vlad URECHE

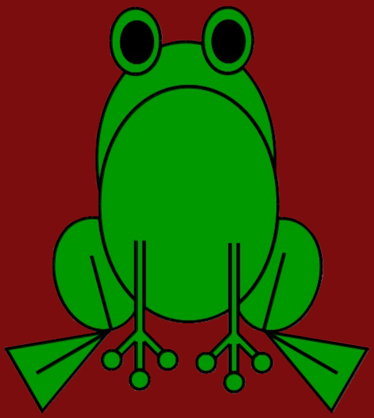
PhD student in the Scala Team @ EPFL

Miniboxing guy. Also worked on specialization, the backend and scaladoc.

@VladUreche

@VladUreche

vlad.ureche@epfl.ch



Late Data Layout:

Unifying Data Representation Transformations

scala-miniboxing.org/ldl

Late Data Layout: Unifying Data Representation Transformations

Vlad Ureche Eugene Burmako Martin Odersky
EPFL, Switzerland
{first.last}@epfl.ch

Abstract

Values need to be represented differently when interacting with certain language features. For example, an integer has to take an object-based representation when interacting with erased generics, although, for performance reasons, the stack-based value representation is better. To abstract over these implementation details, some programming languages choose to expose a unified high-level concept (the integer) and let the compiler choose its exact representation and insert coercions where necessary.

This pattern appears in multiple language features such as value classes, specialization and multi-stage programming: they all expose a unified concept which they later refine into multiple representations. Yet, the underlying compiler implementations typically entangle the core mechanism with assumptions about the alternative representations and their interaction with other language features.

In this paper we present the Late Data Layout mechanism, a simple but versatile type-driven generalization that subsumes and improves the state-of-the-art representation

1. Introduction

Language and compiler designers are well aware of the intricacies of erased generics [15, 21, 30, 32, 35, 42, 46, 75], one of which is requiring object-based representations for primitive types. To illustrate this, let us analyze the `identity` method, parameterized on the argument type, `T`:

```
def identity[T](arg: T): T = arg
val x: Int = identity[Int](5)
```

The low-level compiled code for `identity` needs to handle incoming arguments of different sizes and semantics: booleans, bytes, characters, integers, floating point numbers and references to heap-allocated objects. To implement this, some compilers impose a uniform representation, usually based on references to heap objects. This means that primitive types, such as integers, have to be represented as objects when passed to generic methods. The process of representing primitive types as objects is called boxing. Since boxing slows down execution, whenever primitive types are used outside generic environments, they use their stack-based

OOPSLA 2014, Portland, OR

Late Data Layout: Unifying Data Representation Transformations

Vlad Ureche Eugene Burmako Martin Odersky

EPFL, Switzerland

{first.last}@epfl.ch

Abstract

Values need to be represented differently when interacting with certain language features. For example, an integer has to take an object-based representation when interacting with erased generics, although, for performance reasons, the stack-based value representation is better. To abstract over these implementation details, some programming languages choose to expose a unified high-level concept (the integer) and let the compiler choose its exact representation and insert coercions where necessary.

This pattern appears in multiple language features such as value classes, specialization and multi-stage programming: they all expose a unified concept which they later refine into multiple representations. Yet, the underlying compiler implementations typically entangle the core mechanism with assumptions about the alternative representations and their interaction with other language features.

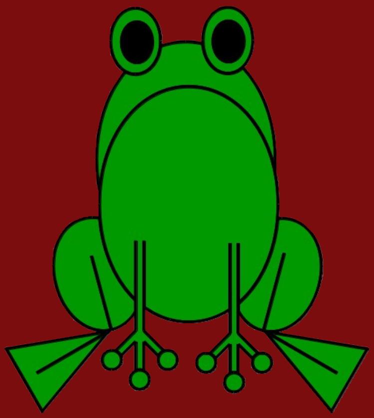
In this paper we present the Late Data Layout mechanism, a simple but versatile type-driven generalization that subsumes and improves the state-of-the-art representation

1. Introduction

Language and compiler designers are well aware of the intricacies of erased generics [15, 21, 30, 32, 35, 42, 46, 75], one of which is requiring object-based representations for primitive types. To illustrate this, let us analyze the `identity` method, parameterized on the argument type, `T`:

```
def identity[T](arg: T): T = arg
val x: Int = identity[Int](5)
```

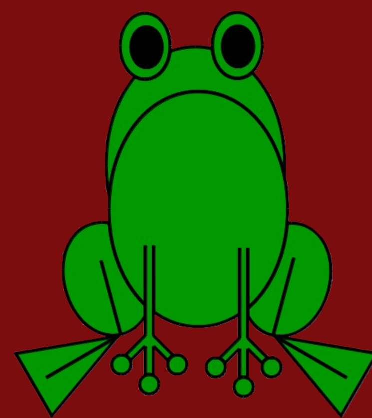
The low-level compiled code for `identity` needs to handle incoming arguments of different sizes and semantics: booleans, bytes, characters, integers, floating point numbers and references to heap-allocated objects. To implement this, some compilers impose a uniform representation, usually based on references to heap objects. This means that primitive types, such as integers, have to be represented as objects when passed to generic methods. The process of representing primitive types as objects is called boxing. Since boxing slows down execution, whenever primitive types are used outside generic environments, they use their stack-based



Late Data Layout:

Unifying Data Representation Transformations

scala-miniboxing.org/ldl

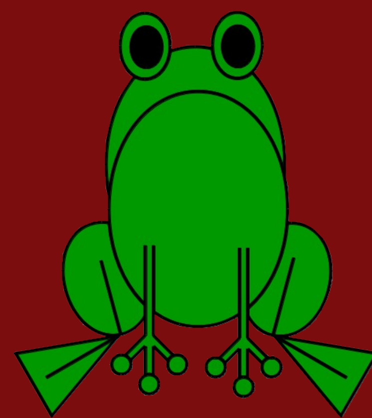


Late Data Layout:

Unifying Data Representation Transformations

- **compiler** transformations
- separate compilation
- **global scope**

scala-miniboxing.org/ldl

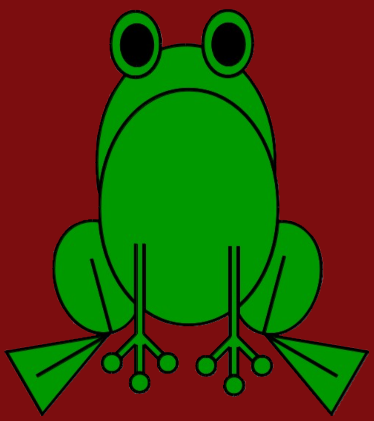


Late Data Layout:

Unifying Data Representation Transformations

- unboxing, value classes
- **how data is represented**

scala-miniboxing.org/ldl



Late Data Layout:

Unifying Data Representation Transformations

- what is there to unify?
- **why bother?**

Motivation

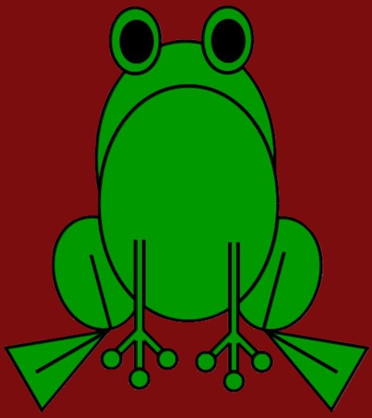
Transformation

Conclusion

scala-miniboxing.org/ldl

- Representation Transformations

- Representation Transformations
  - Unboxing Primitive Types

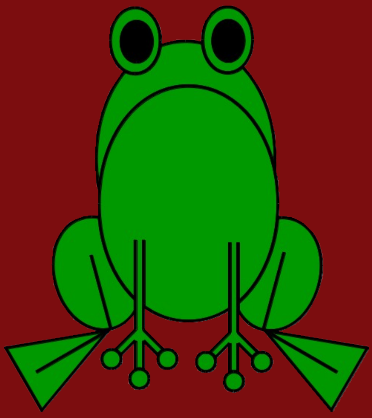


Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)

Representation Transformations

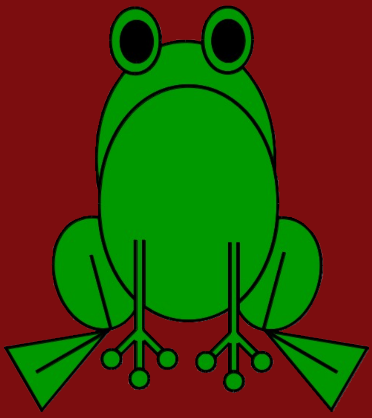
- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes



Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes

motivated by
erased generics

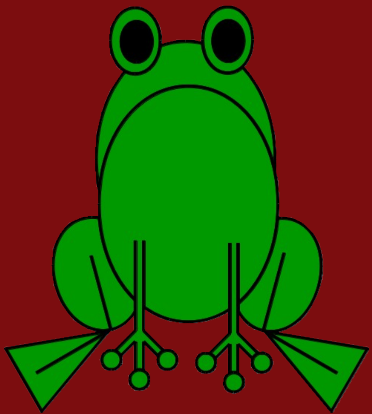


scala-miniboxing.org/ldl

Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)

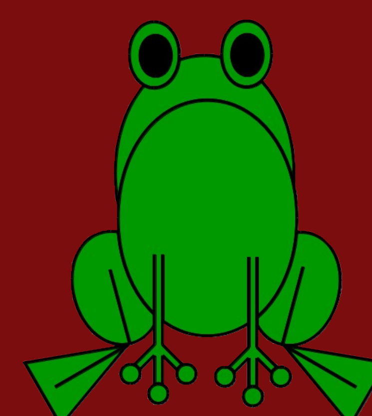
motivated by erased generics



Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)
- Function Representation

motivated by
erased generics

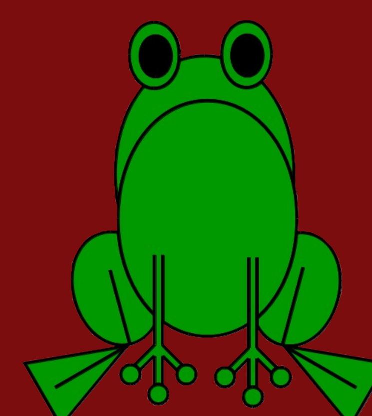


scala-miniboxing.org/ldl

Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)
- Function Representation

motivated by erased generics



scala-miniboxing.org/ldl

Unboxing Primitive Types

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

Unboxing Primitive Types

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

$\Big\downarrow$ scalac / javac

Unboxing Primitive Types

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

scalac / javac

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

Unboxing Primitive Types

identity(5)

Unboxing Primitive Types

identity(5)

scalac / javac

Unboxing Primitive Types

identity(5)

scalac / javac

identity(j.l.Integer.valueOf(**5**)**)**.intValue

Unboxing Primitive Types

identity(5)

scalac / javac

identity(j.l.Integer.valueOf(**5**)**)**.intValue

Object representation

Unboxing Primitive Types

identity(5)

scalac / javac

inflates heap requirements

produces garbage

identity(j.l.Integer.valueOf(5)).intValue

Object representation

indirect (slow) access to the value

breaks locality guarantees

Unboxing Primitive Types

```
val five: Int = 5
```

Unboxing Primitive Types

scala.Int behaves like an object
(has methods, can be used with generics)

```
val five: Int = 5
```

Unboxing Primitive Types

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5

scalac

Unboxing Primitive Types

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5

scalac

val five: int = 5

Unboxing Primitive Types

scala.Int behaves like an object
(has methods, can be used with generics)

val five: Int = 5

scalac

val five: int = 5

Unboxed integer

Unboxing Primitive Types

```
val five: Int = identity(5)
```

Unboxing Primitive Types

```
val five: Int = identity(5)
```

scalac

Unboxing Primitive Types

val five: Int = identity(5)

scalac

val five: int =

Unboxing Primitive Types

```
val five: Int = identity(5)
```

scalac

```
val five: int =
  identity(I.valueOf(5)).intValue
```

Unboxing Primitive Types

```
val five: Int = identity(5)
```

scalac

```
val five: int =
  identity(I.valueOf(5)).intValue
```

Boxing coercion

Unboxing Primitive Types

```
val five: Int = identity(5)
```

scalac

```
val five: int =
  identity(I.valueOf(5)).intValue
```

Boxing coercion

Unboxing coercion

Unboxing Primitive Types

scala.Int

Unboxing Primitive Types

$$\overbrace{\hspace{20em}}^{\textbf{scala.Int}}$$

Unboxing Primitive Types

scala.Int

int

- fast access
- no garbage collection
- locality

Unboxing Primitive Types

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

Unboxing Primitive Types

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

incompatible
→ **coercions**

Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)
- Function Representation

scala-miniboxing.org/ldl

Specialization

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

Specialization

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

specialization

Specialization

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

specialization

def identity(t: Object): Object = t

Specialization

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

specialization

```
def identity(t: Object): Object = t
def identity_Z(t: bool): bool = t
```

Specialization

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

specialization

```
def identity(t: Object): Object = t
def identity_Z(t: bool): bool = t
def identity_C(t: char): char = t
```

Specialization

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

specialization

```
def identity(t: Object): Object = t
def identity_Z(t: bool): bool = t
def identity_C(t: char): char = t
... (7 other variants)
```

Specialization

identity(5)

Specialization

identity(5)

specialization

Specialization

identity(5)

specialization

identity_I(5)

Specialization

identity(5)

specialization

identity_I(5)

The variant of identity specialized for **int**

Specialization

identity(5)

specialization

identity_I(5) **// no boxing!**

The variant of identity
specialized for **int**

Specialization

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

Specialization

def tupled[T1, T2](t1: T1, t2: T2) ...

↓ specialization

Specialization

def tupled[T1, T2](t1: T1, t2: T2) ...

specialization

// 100 methods (10^2)

Specialization

def tupled[T1, T2](t1: T1, t2: T2) ...

specialization

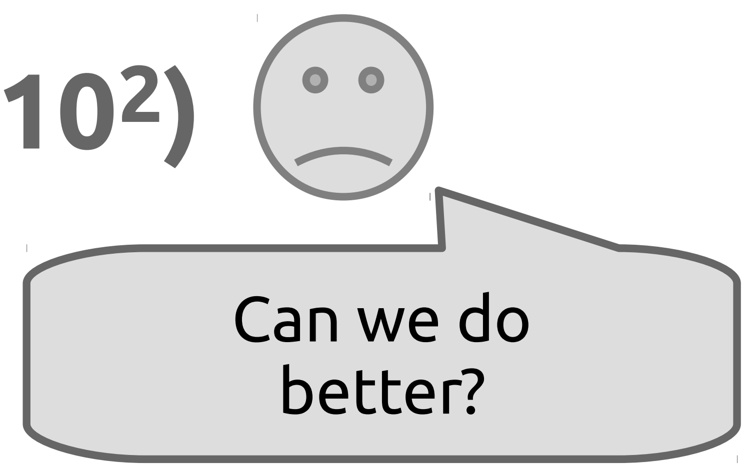
// 100 methods (10^2) 

Specialization

def tupled[T1, T2](t1: T1, t2: T2) ...

↓ specialization

// 100 methods (10^2)

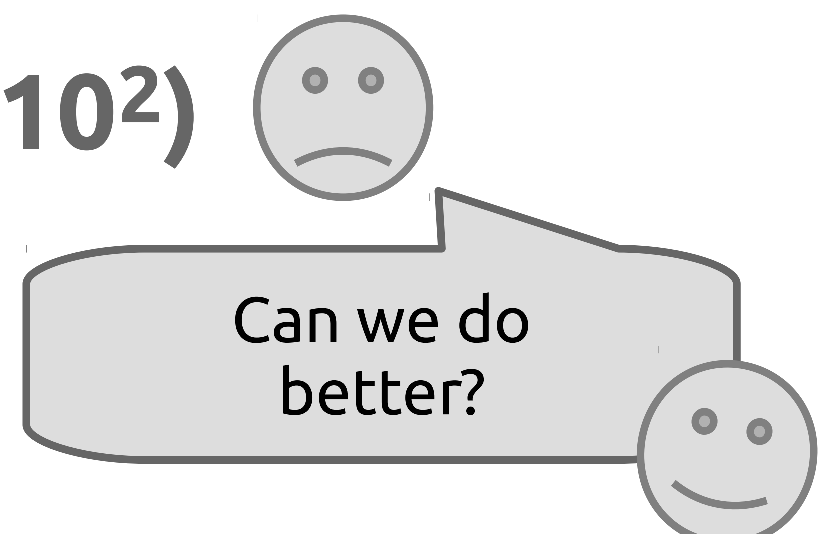


Specialization

def tupled[T1, T2](t1: T1, t2: T2) ...

specialization

// 100 methods (10^2)



Miniboxing

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

Miniboxing

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

miniboxing

Miniboxing

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

miniboxing

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

Miniboxing

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

miniboxing

```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
```

Miniboxing

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

miniboxing

```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
```

long **encodes** all primitive types

Miniboxing

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

miniboxing

```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
```

long **encodes** all primitive types

Only 2^n variants

scala-miniboxing.org/ldl

Miniboxing

identity(3)

Miniboxing

identity(3)

$\Big\downarrow$ miniboxing

Miniboxing

identity(3)

miniboxing

identity_M(..., int2minibox(3))

Miniboxing

identity(3)

miniboxing

identity_M(..., int2minibox(3))

The miniboxed
variant of identity

Miniboxing

identity(3)

miniboxing

identity_M(..., int2minibox(3))

The miniboxed variant of identity

Coercion

Miniboxing

T

Miniboxing

T

Miniboxing

T

long

- preferred encoding

Miniboxing

T

long

- preferred encoding

T (erased to Object)

- fallback encoding
- **compatible with**
  - **virtual dispatch**
  - **subtyping**
  - **erased generics**

Miniboxing

T

long

- preferred encoding

T (erased to Object)

- fallback encoding
- **compatible with**
  - **virtual dispatch**
  - **subtyping**
  - **erased generics**

incompatible
→ **coercions**

Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)
- Function Representation

Value Classes

```
def abs(c: Complex): Double = ...
```

Value Classes

```
def abs(c: Complex): Double = ...
```

value class transformation

Value Classes

```
def abs(c: Complex): Double = ...
```

value class transformation

```
def abs(c_re: Double,
        c_im: Double): Double = ...
```

Value Classes

```
def abs(c: Complex): Double = ...
```

value class transformation

```
def abs(c_re: Double,
        c_im: Double): Double = ...
```

No object created!

Value Classes

```
val c: Complex = Complex(2,1)
```

Value Classes

```
val c: Complex = Complex(2,1)
```

value class transformation

Value Classes

val c: Complex = Complex(2,1)

value class transformation

val c_re: Double = 2
val c_im: Double = 1

Value Classes

```
val c: Complex = Complex(2,1)
```

value class transformation

```
val c_re: Double = 2
val c_im: Double = 1
```

No object created!

Value Classes

```
val a: Any = c
```

Value Classes

```
val a: Any = c
```

value class transformation

Value Classes

```
val a: Any = c
```

value class transformation

```
val a: Any =
        new Complex(c_re, c_im)
```

Value Classes

```
val a: Any = c
```

value class transformation

```
val a: Any =
        new Complex(c_re, c_im)
```

Coercion!

Value Classes

value class

Value Classes

value class

Value Classes

value class

structure **(by-val)**

- preferred encoding

Value Classes

value class

structure **(by-val)**

- preferred encoding

class **(by-ref)**

- fallback encoding
- **compatible with**
  - **subtyping**
  - **erased generics**

Value Classes

value class

structure **(by-val)**

- preferred encoding

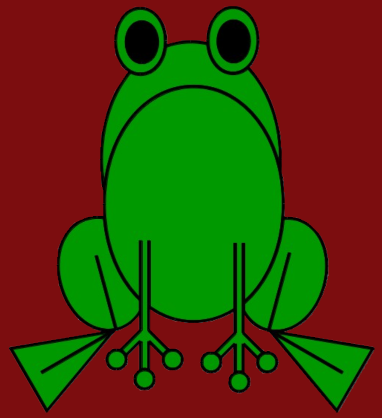
class **(by-ref)**

- fallback encoding
- **compatible with**
  - **subtyping**
  - **erased generics**

incompatible
→ **coercions**

Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)
- Function Representation



Staging

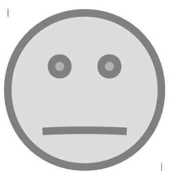
T

Staging

T

Staging

T



T (direct)

- executed in this stage
- stores **a value**

Staging

T

T (direct)

- executed in this stage
- stores **a value**

Rep[T] (lifted)

- executed in the next stage
- stores **the expression** that produces the value

Staging

T

T (direct)

- executed in this stage
- stores **a value**

Rep[T] (lifted)

- executed in the next stage
- stores **the expression** that produces the value

incompatible
→ **coercions**

Representation Transformations

- Unboxing Primitive Types
- Specialization (Miniboxing)
- Value Classes
- Staging (Multi-Stage Programming)
- Function Representation

Function Representation

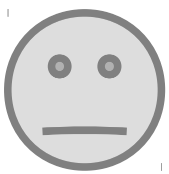
scala.FunctionX

Function Representation

scala.FunctionX

Function Representation

scala.FunctionX

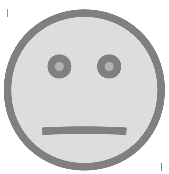


FunctionX

- **compatible with the library**
- does not have all specializations
- slow when used by miniboxed code

Function Representation

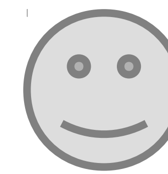
scala.FunctionX



FunctionX

- **compatible with the library**
- does not have all specializations
- slow when used by miniboxed code

MbFunctionX



- fast calling from miniboxed code
- all specializations are there

Function Representation

scala.FunctionX

FunctionX

- **compatible with the library**
- does not have all specializations
- slow when used by miniboxed code

MbFunctionX

- fast calling from miniboxed code
- all specializations are there

incompatible
→ **coercions**

~~Motivation~~

Transformation

Conclusion

~~Motivation~~

Transformation

Conclusion

Late Data Layout (LDL)

Late Data Layout (LDL)

concept

Late Data Layout (LDL)

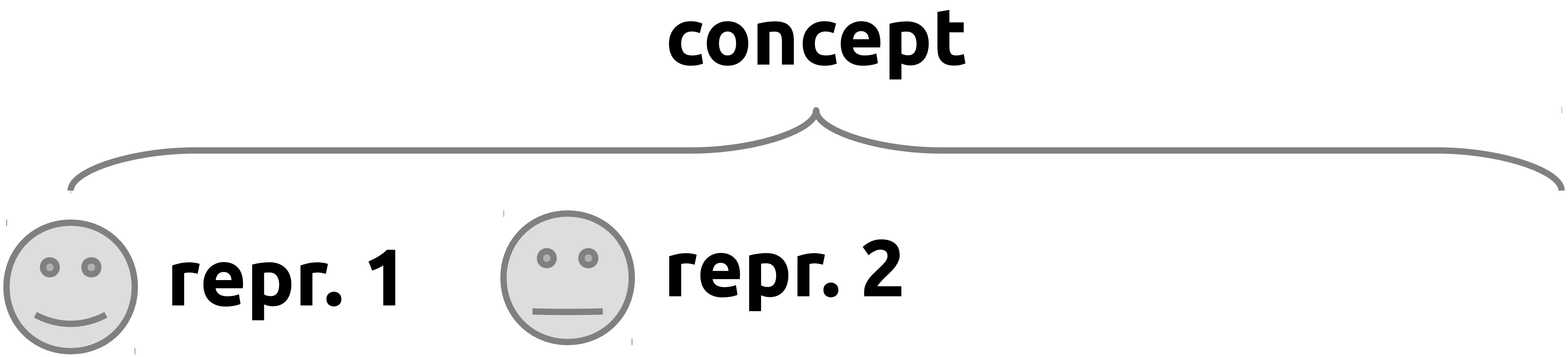
concept

Late Data Layout (LDL)

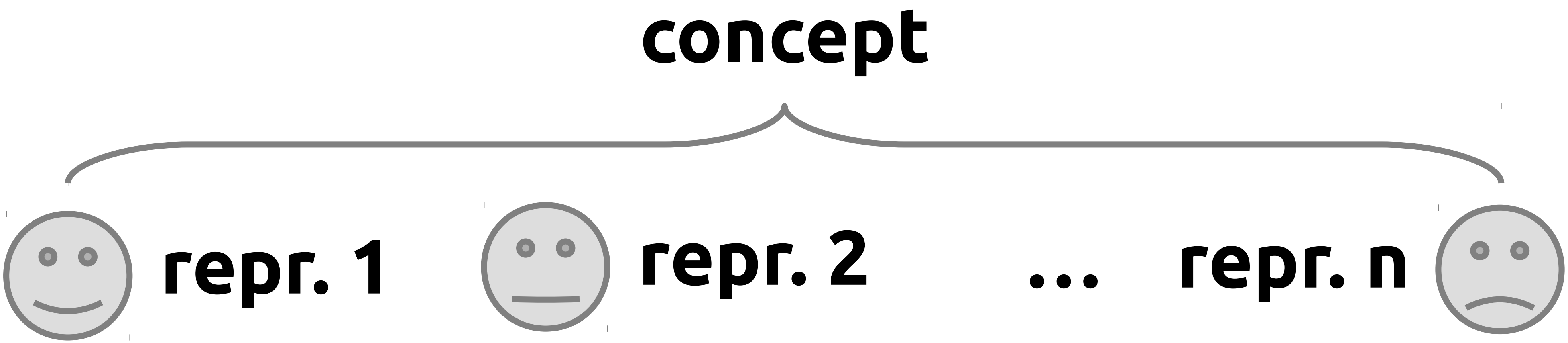
concept

repr. 1

Late Data Layout (LDL)

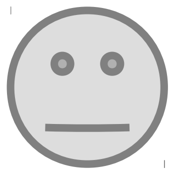
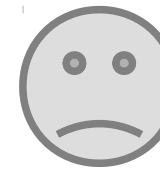


Late Data Layout (LDL)



Late Data Layout (LDL)

concept

 **repr. 1**  **repr. 2** ... **repr. n** 

Constraints from the interaction with other language features:

- generics
- subtyping
- virtual dispatch
- DSL semantics (staging)

Late Data Layout (LDL)

concept

repr. 1 **repr. 2** ... **repr. n**

Constraints from the interaction with other language features:

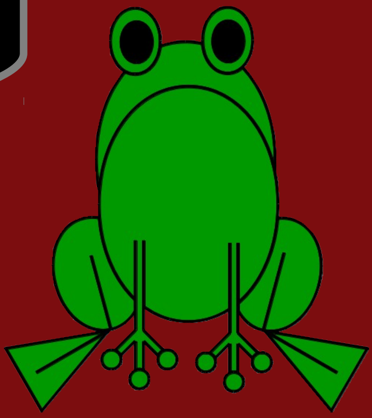
- generics
- subtyping
- virtual dispatch
- DSL semantics (staging)

incompatible
→ **coercions**

How to transform a program?

How to transform a program?

We'll use primitive unboxing
as the running example,
to keep things simple



Unboxing Primitive Types

scala.Int

int

- fast access
- no garbage collection
- locality

java.lang.Integer

- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

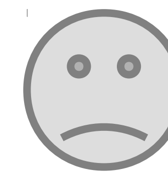
Unboxing Primitive Types

scala.Int

int

- fast access
- no garbage collection
- locality

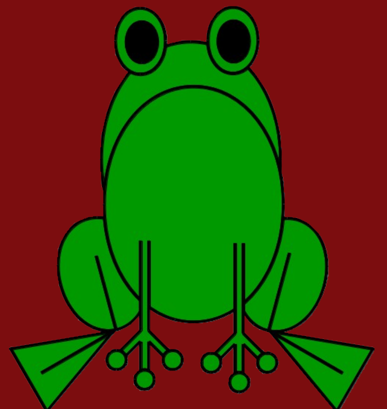
java.lang.Integer



- indirect access
- object allocation
  - and thus garbage collection
- no locality guarantees
- **compatible with erased generics**

incompatible
→ **coercions**

- Naive transformation
- Syntax-based transformation
- Type-based LDL transformation



scala-miniboxing.org/ldl

Naive transformation

```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

naive unboxing

Naive transformation

```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

naive unboxing

```
val x: int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

Naive transformation

```
val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

naive unboxing

representation mismatch:
expected: **int** (unboxed)
found: **Int** (boxed)

```
val x: int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)
```

Naive transformation

val x: Int = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)

naive unboxing

representation mismatch:
expected: **int** (unboxed)
found: **Int** (boxed)

val x: $\boxed{\text{int}}$ = List[Int](1, 2, 3).head
val y: List[Int] = List[Int](x)

representation mismatch:
expected: **Int** (boxed)
found: **int** (unboxed)

Naive transformation

- naively replacing representations
  - leads to mismatches
  - which are hard to recover
    (impossible for value classes and miniboxing)
- we need **coercions** between representations

- Naive transformation
- **Syntax-based transformation**
- Type-based LDL transformation

Syntax-based

- when transforming a value
  - coerce the definition right-hand side
  - coerce all references to it

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

syntax-based unboxing

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

syntax-based unboxing

```
val x: int =
```

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
```

Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
```

There are no references to x,
so there's nothing else to do.

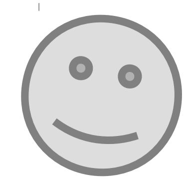
Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
```

syntax-based unboxing

```
val x: int =
    unbox(List[Int](1, 2, 3).head)
```

There are no references to x,
so there's nothing else to do.



Syntax-based

```
val x: Int = List[Int](1, 2, 3).head
val z: Int = x
```

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val z: Int = x
```

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val z: Int = x
```

syntax-based unboxing

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val z: Int = x
```

syntax-based unboxing

```
val x: int =
    unbox(List[Int](1, 2, 3).head)
```

Syntax-based

Transform one by one

```
val x: Int = List[Int](1, 2, 3).head
val z: Int = x
```

syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: Int = box(x)
```

Syntax-based

```
val x: int =
        unbox(List[Int](1, 2, 3).head)
val z: Int = box(x)
```

Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: Int = box(x)
```

syntax-based unboxing

Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: Int = box(x)
```

syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: int =
```

Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: Int = box(x)
```

syntax-based unboxing

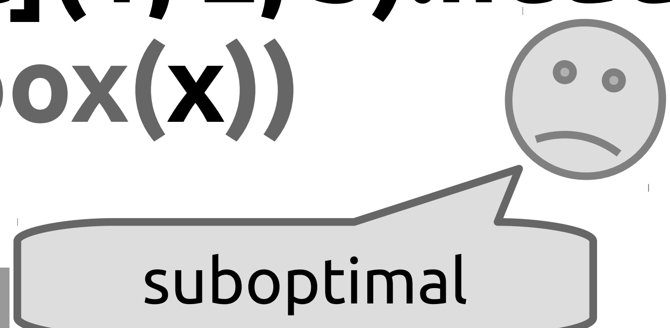
```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: int = unbox(box(x))
```

Syntax-based

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: Int = box(x)
```

syntax-based unboxing

```
val x: int =
      unbox(List[Int](1, 2, 3).head)
val z: int = unbox(box(x))
```



Peephole Optimization

```
val z: int = unbox(box(x))
```

Peephole Optimization

val z: int = unbox(box(x))

peephole

Peephole Optimization

```
val z: int = unbox(box(x))
```

peephole

```
val z: int = x
```

Peephole Optimization

val z: int = unbox(box(x))

peephole

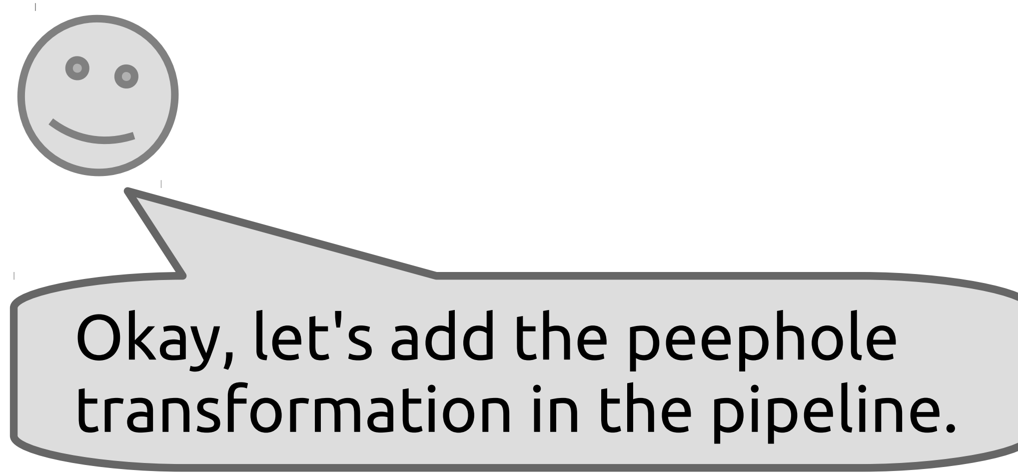
val z: int = x

Peephole Optimization

```
val z: int = unbox(box(x))
```

peephole

```
val z: int = x
```



Syntax-based

```
def choice(t1: Int, t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Syntax-based

Transform one by one

```
def choice(t1: Int, t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Syntax-based

```
def choice(t1: int, t2: Int): Int =
  if (Random.nextBoolean())
    box(t1)
  else
    t2
```

Syntax-based

```
def choice(t1: int, t2: int): Int =
  if (Random.nextBoolean())
    box(t1)
  else
    box(t2)
```

Syntax-based

```
def choice(t1: int, t2: int): Int =
  if (Random.nextBoolean())
    box(t1)
  else
    box(t2)
```

Anything missing?

Syntax-based

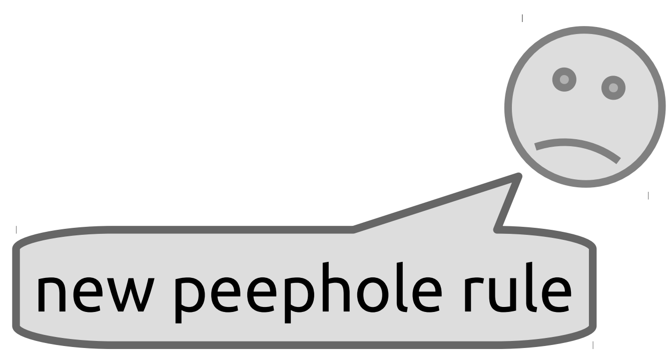
```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```

Syntax-based

```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```

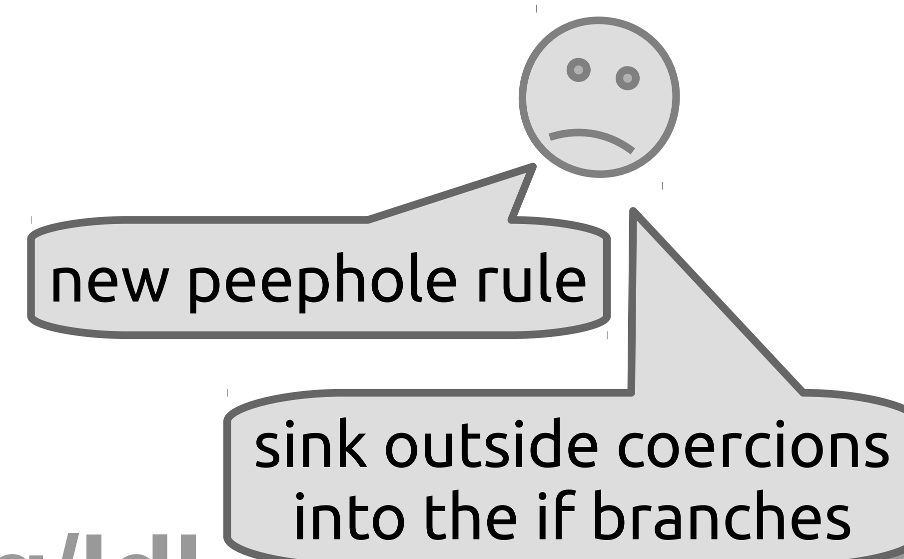
Syntax-based

```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```



Syntax-based

```
def choice(t1: int, t2: int): int =
  unbox(if (Random.nextBoolean())
    box(t1)
  else
    box(t2))
```



Syntax-based

```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    unbox(box(t1))
  else
    unbox(box(t2))
```

Syntax-based

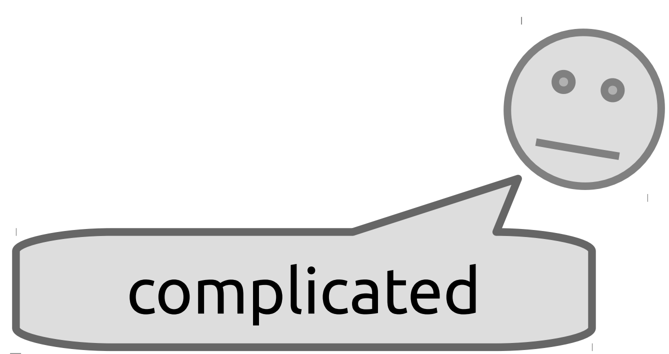
```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    unbox(box(t1))
  else
    unbox(box(t2))
```

Syntax-based

```scala
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Syntax-based

```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



Syntax-based

- peephole transformation does not scale
  - needs **multiple rules** for each node
  - needs **stateful rewrites**
  - leads to an explosion of rules **x** states

Details in
the paper

- Naive transformation
- Syntax-based transformation
- Type-based LDL transformation

LDL Transformation

- propagate representation information
  - into the type system (based on annotated types)
  - allows **selective marking** of values to be unboxed

LDL Transformation

- re-typecheck the tree
  - exposes inconsistencies in the representation
    - based on backward type propagation
    - from local type inference
  - so we introduce coercions
    - **optimally**, only when representations don't match

LDL Transformation

- three-stage mechanism
  - inject → annotate the values to be unboxed
  - coerce → introduce coercion markers
  - commit → commit to the alternative representations

LDL Transformation

Warning!

Throughout the presentation we'll be writing annotations written **before types** (e.g. “@unboxed Int”), although in the Scala syntax they are written **after the type** (e.g.“Int @unboxed”). This makes it easier to read the types aloud.

LDL Transformation

inject
coerce
commit

LDL Transformation

- inject
- coerce
- commit

```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

- **inject**
- coerce
- commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

configurable introduction of annotations based on external constraints

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

the return type of choice
is @unboxed Int

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

the return type of choice
is @unboxed Int

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

the return type of choice
is @unboxed Int

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                              : @unboxed Int
```

LDL Transformation

inject
coerce
commit

the return type of choice
is **@unboxed Int**

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                              : @unboxed Int
```

expected type
(part of local type inference)

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                       : @unboxed Int
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean()) : Boolean
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean()) : Boolean
    t1
  else
    t2
```

matches:
expected: **Boolean**
found: **Boolean**

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
   t1  : @unboxed Int
  else
   t2
```

matches:
expected: @unboxed Int
found: @unboxed Int

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2 : @unboxed Int
```

LDL Transformation

- inject
- **coerce**
- commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2 : @unboxed Int
```

matches:

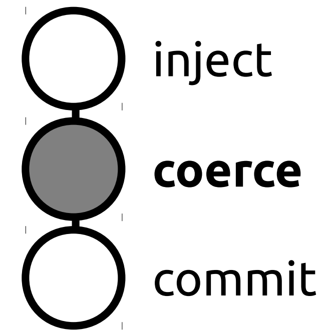
...

LDL Transformation

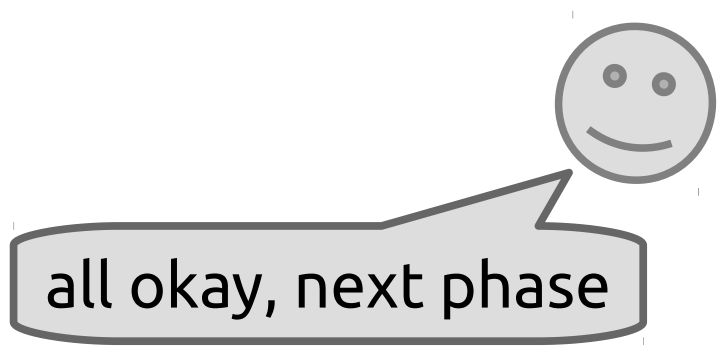
inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation




```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Replace:
@unboxed Int → **int**
(not showing **Int** → **j.l.Integer**)

LDL Transformation

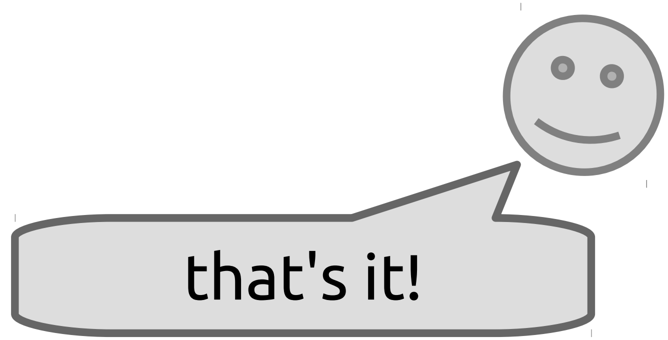
inject
coerce
commit

```
def choice(t1: int,
           t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

LDL Transformation

inject
coerce
commit

```
def choice(t1: int,
                t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



LDL Transformation

- three-stage mechanism
  - inject: add annotations
  - coerce: add coercions (based on the annotations)
  - commit: final representation semantics

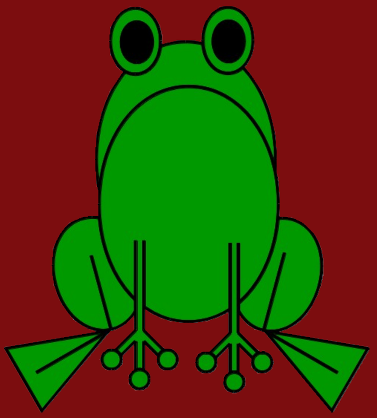
LDL Transformation

- Scalac's erasure
  - similar transformation
  - less flexible (no annotations)
  - entangled with other transformations
- we took what's good
  - and allowed the transformation to work on other use cases as well

Type-based LDL transformation

Properties

- Consistency
- Selectivity
- Optimality (not formally proven yet)



Consistency

inject

coerce

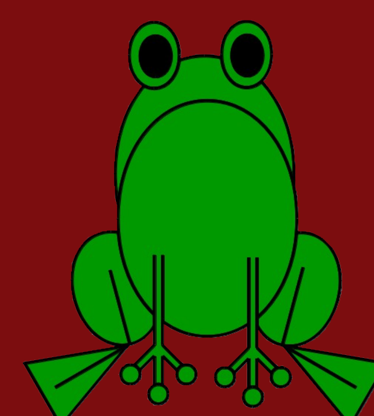
commit

- representations become explicit in types
  - representation mismatches
    - become type mismatches
    - are exposed by the type system
  - mismatches lead to coercions
    - explicit bridges between representations
    - are introduced automatically
  - regardless of the representations
    - at a meta-level

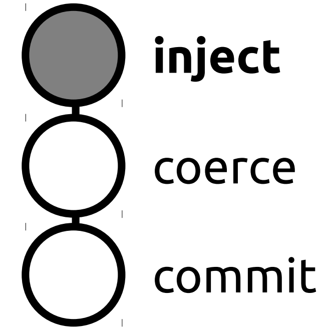
Type-based LDL transformation

Properties

- Consistency
- Selectivity
- Optimality (not formally proven yet)



Selectivity



- annotations allow selectively picking the values to be transformed
  - value classes
    - cannot unbox multi-param values in return position (not supported by the JVM platform)
    - bridge methods
  - staging
    - annotations signal **domain-specific knowledge**
      - can occur inside generics (List[@staged Int])

Selectivity

inject
coerce
commit

```scala
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Selectivity

- **inject**
- coerce
- commit

what if we did not annotate t1?

```
def choice(t1: Int,
           t2: Int): Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Selectivity

- **inject**
- coerce
- commit

what if we did not annotate t1?

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```

Selectivity

inject
coerce
commit

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2                                      : @unboxed Int
```

Selectivity

inject
coerce
commit

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
   t1 : @unboxed Int
  else
   t2
```

Selectivity

inject
coerce
commit

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

mismatch:
expected: @unboxed Int
found: Int

Selectivity

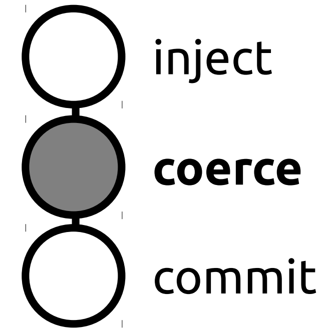
inject
coerce
commit

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1 : @unboxed Int
  else
    t2
```

mismatch:
expected: @unboxed Int
found: Int

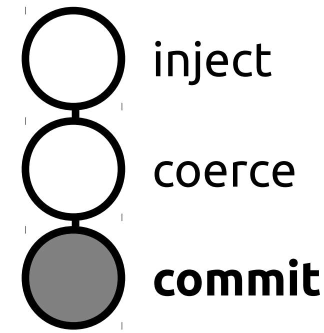
coercion

Selectivity



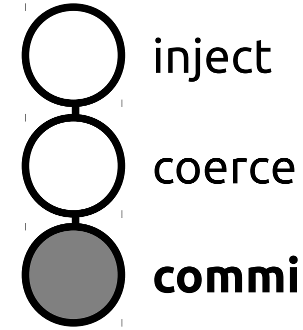

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Selectivity




```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Selectivity

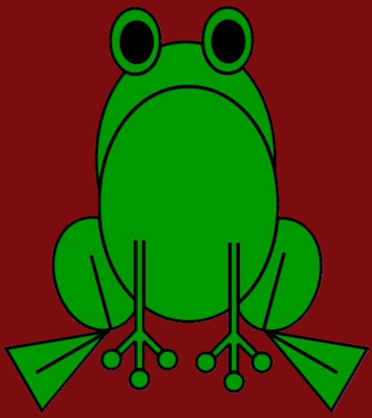



```
def choice(t1: Int,
           t2: int): int =
  if (Random.nextBoolean())
    t1.intValue
  else
    t2
```

Type-based LDL transformation

Properties

- Consistency
- Selectivity
- Optimality (not formally proven yet)



scala-miniboxing.org/ldl

Optimality

inject
coerce
commit

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Optimality

inject
coerce
commit

```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Optimality

inject
coerce
commit

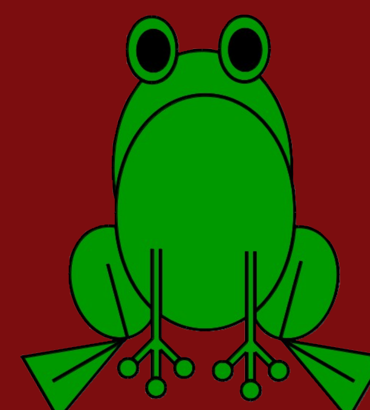
```
def choice(t1: Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    unbox(t1)
  else
    t2
```

Coercions are sunk in the tree →
excute only if necessary

~~Motivation~~

~~Transformation~~

Conclusion



scala-miniboxing.org/ldl

~~Motivation~~

~~Transformation~~

Conclusion

Conclusion

LDL is a

- compile-time transformation
  - compatible with **separate compilation**
  - compatible with **partial transformation**

Conclusion

LDL is a

- compile-time transformation
  - compatible with **separate compilation**
  - compatible with **partial transformation**
  - **global scope** (Graal/Truffle: local scope)
    - optimizes all data in a program

Conclusion

LDL is a

- compile-time transformation
  - compatible with **separate compilation**
  - compatible with **partial transformation**
  - **global scope** (Graal/Truffle: local scope)
    - optimizes all data in a program / containers
  - **conservative** (Graal/Truffle: speculative)

Conclusion

LDL is a

- compile-time transformation
  - compatible with **separate compilation**
  - compatible with **partial transformation**
  - **global scope** (Graal/Truffle: local scope)
    - optimizes all data in a program / containers
  - **conservative** (Graal/Truffle: speculative)

Complementary optimizations

Conclusion

LDL allows

- splitting a high-level concept
  - into multiple representations
  - in a **consistent** way (through coercions)
  - in a **selective** way (through annotations)
  - in an **optimal** way (coerce only if necessary)

Conclusion

LDL is used in

- the miniboxing plugin
  - for specialization
  - for function representation
- other prototypes
  - value classes
  - staging

Conclusion
LDL is used in

- the miniboxing plugin
  - for specialization
  - for function representation
- other prototypes
  - value classes
  - staging

Don't take my word for it

Conclusion

LDL is used in

- the miniboxing plugin
  - for specialization
  - for function representation
- other prototypes
  - value classes
  - staging

Don't take my word for it

Sources on github, artifact online.

Late Data Layout: Unifying Data Representation Transformations

Vlad Ureche Eugene Burmako Martin Odersky
EPFL, Switzerland
{first.last}@epfl.ch

Abstract

Values need to be represented differently when interacting with certain language features. For example, an integer has to take an object-based representation when interacting with erased generics, although, for performance reasons, the stack-based value representation is better. To abstract over these implementation details, some programming languages choose to expose a unified high-level concept (the integer) and let the compiler choose its exact representation and insert coercions where necessary.

This pattern appears in multiple language features such as value classes, specialization and multi-stage programming: they all expose a unified concept which they later refine into multiple representations. Yet, the underlying compiler implementations typically entangle the core mechanism with assumptions about the alternative representations and their interaction with other language features.

In this paper we present the Late Data Layout mechanism, a simple but versatile type-driven generalization that subsumes and improves the state-of-the-art representation

1. Introduction

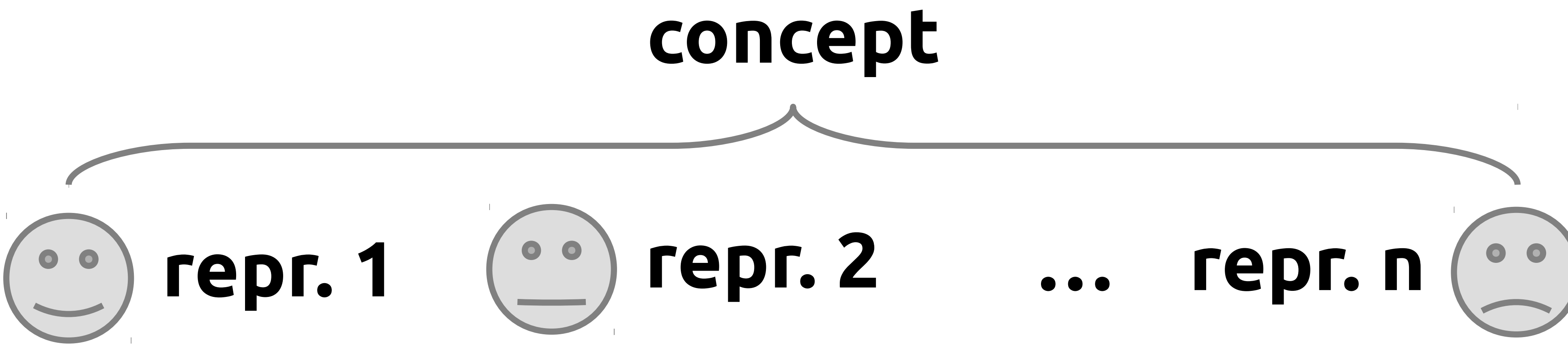
Language and compiler designers are well aware of the intricacies of erased generics [15, 21, 30, 32, 35, 42, 46, 75], one of which is requiring object-based representations for primitive types. To illustrate this, let us analyze the `identity` method, parameterized on the argument type, `T`:

```
def identity[T](arg: T): T = arg
val x: Int = identity[Int](5)
```

The low-level compiled code for `identity` needs to handle incoming arguments of different sizes and semantics: booleans, bytes, characters, integers, floating point numbers and references to heap-allocated objects. To implement this, some compilers impose a uniform representation, usually based on references to heap objects. This means that primitive types, such as integers, have to be represented as objects when passed to generic methods. The process of representing primitive types as objects is called boxing. Since boxing slows down execution, whenever primitive types are used outside generic environments, they use their stack-based

Conclusion

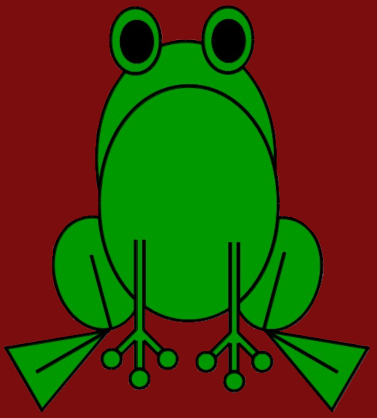
What's your use-case?



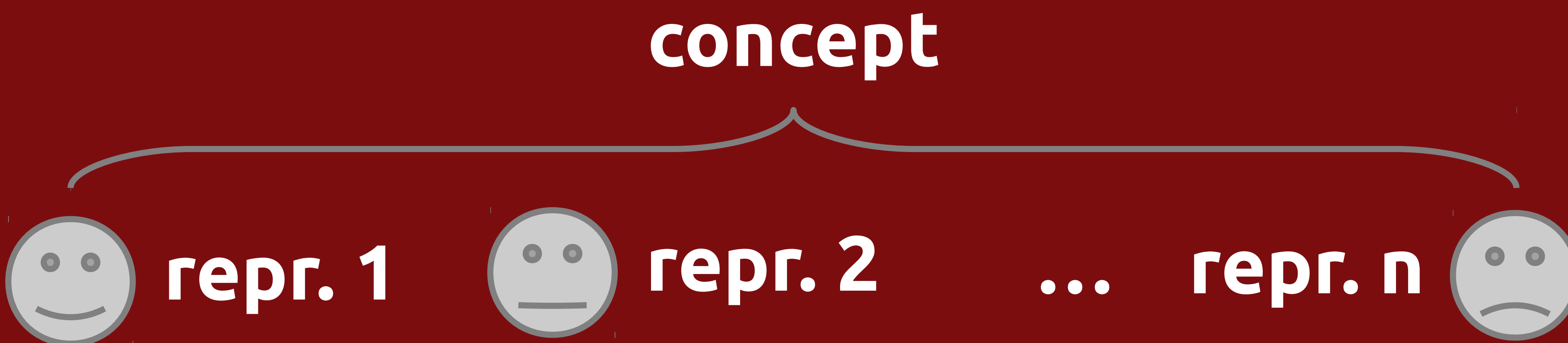
Credits and Thank you-s

- Cristian Talau - developed the initial prototype, as a semester project
- Eugene Burmako - the value class plugin based on the LDL transformation
- Aymeric Genet - developing collection-like benchmarks for the miniboxing plugin
- Martin Odersky, for his patient guidance
- Eugene Burmako, for trusting the idea enough to develop the value-plugin based on the LDL transformation
- Iulian Dragos, for his work on specialization and many explanations
- Miguel Garcia, for his original insights that spawned the miniboxing idea
- Michel Schinz, for his wonderful comments and enlightening ACC course
- Andrew Myers and Roland Ducournau for the discussions we had and the feedback provided
- Heather Miller for the eye-opening discussions we had
- Vojin Jovanovic, Sandro Stucki, Manohar Jonalagedda and the whole LAMP laboratory in EPFL for the extraordinary atmosphere
- Adriaan Moors, for the miniboxing name which stuck :))
- Thierry Coppey, Vera Salvisberg and George Nithin, who patiently listened to many presentations and provided valuable feedback
- Grzegorz Kossakowski, for the many brainstorming sessions on specialization
- Erik Osheim, Tom Switzer and Rex Kerr for their guidance on the Scala community side
- OOPSLA paper and artifact reviewers, who reshaped the paper with their feedback
- Sandro, Vojin, Nada, Heather, Manohar - reviews and discussions on the LDL paper
- Hubert Plociniczak for the type notation in the LDL paper
- Denys Shabalin, Dmitry Petrashko for their patient reviews of the LDL paper

Special thanks to the Scala Community for their support!
(@StuHood, @vpatryshev and everyone else!)



scala-miniboxing.org/ldl



scala-miniboxing.org/ldl