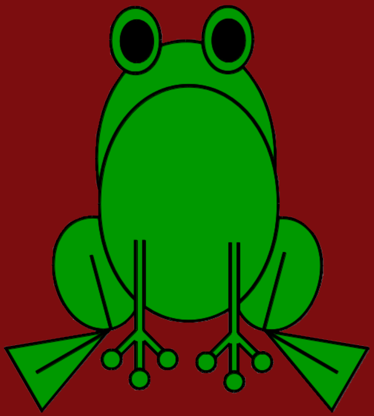


Late Data Layout:

Unifying Data Representation Transformations

Vlad Ureche   Eugene Burmako   Martin Odersky
École polytechnique fédérale de Lausanne, Switzerland
{first.last}@epfl.ch

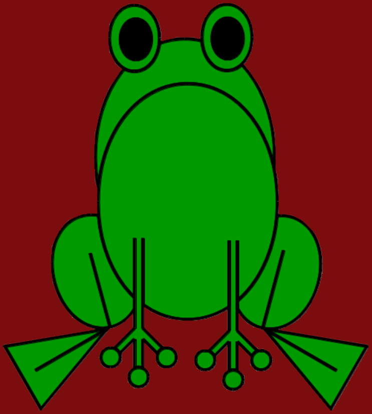
23rd of October 2014
OOPSLA '14
Portland, OR

scala-ldl.org

Late Data Layout:

Unifying Data Representation Transformations

scala-ldl.org

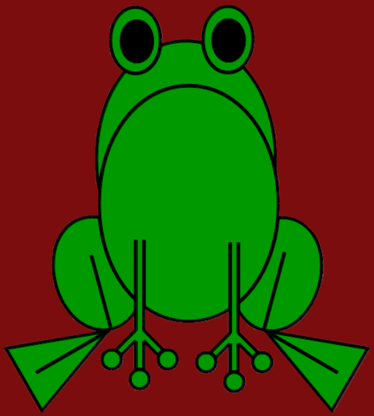


Late Data Layout:

Unifying Data Representation Transformations

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

scala-ldl.org

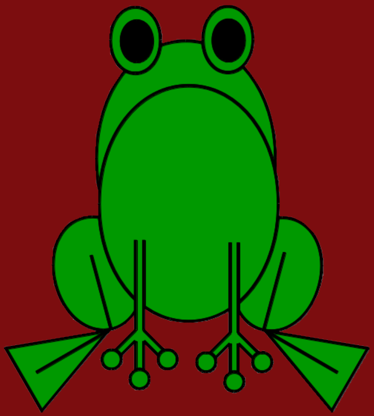


Late Data Layout:

Unifying $\underbrace{\text{Data Representation}}$ Transformations

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

scala-ldl.org



Late Data Layout:

$\underbrace{\text{Unifying}}$ Data Representation Transformations

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

scala-ldl.org

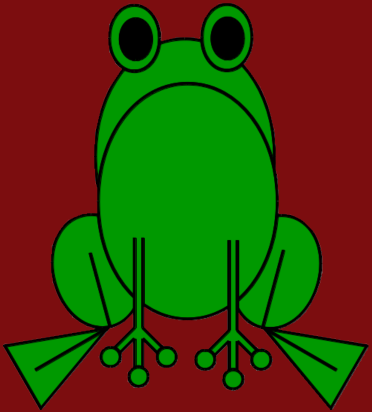
Motivation

Transformation

Properties

Benchmarks

Conclusion



○ Representation Transformations

- Representation Transformations
  - Unboxing Primitive Types

Unboxing Primitive Types

Unboxing Primitive Types

int

- value
- no garbage collection
- locality

Unboxing Primitive Types

int

- value
- 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

int

- value
- no garbage collection
- locality

in Java, **programmers are responsible for the choice of representation**

java.lang.Integer

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

Unboxing Primitive Types

int

- value
- no garbage collection
- locality

java.lang.Integer

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

in Java, **programmers are responsible for the choice of representation**

What about Scala?

Unboxing Primitive Types

int

- value
- 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

int

- value
- 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

- value
- 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

$\overbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

🙂 **int**

- value
- no garbage collection
- locality

java.lang.Integer 🙁

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

scalac {

Unboxing Primitive Types

$$\underbrace{\text{int} \qquad\qquad\qquad \text{java.lang.Integer}}_{\text{}}$$

scala.Int

☺ **int**

- value
- no garbage collection
- locality

java.lang.Integer ☹

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

scalac { ① Choice of representation

Unboxing Primitive Types

scala.Int

int

- value
- no garbage collection
- locality

java.lang.Integer

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

scalac
① Choice of representation
② Coercions between representations

Representation Transformations

- Unboxing Primitive Types
- Value Classes

Value Classes

Value Classes

value class

Value Classes

value class

struct **(by-val)**

- preferred encoding
- fields are inlined
- no heap allocations

Value Classes

value class

struct **(by-val)**

- preferred encoding
- fields are inlined
- no heap allocations

object **(by-ref)**

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

Value Classes

value class

$$\underbrace{\hspace{10cm}}$$

🙂 **struct (by-val)**

- preferred encoding
- fields are inlined
- no heap allocations

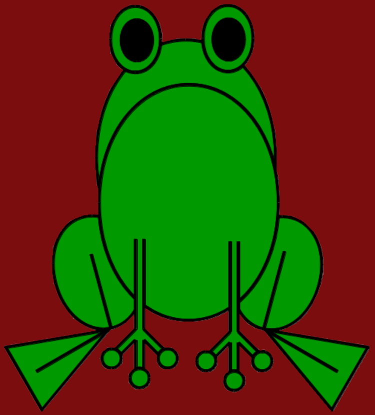
object (by-ref) 🙁

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

scalac {
① Choice of representation
② Coercions between representations

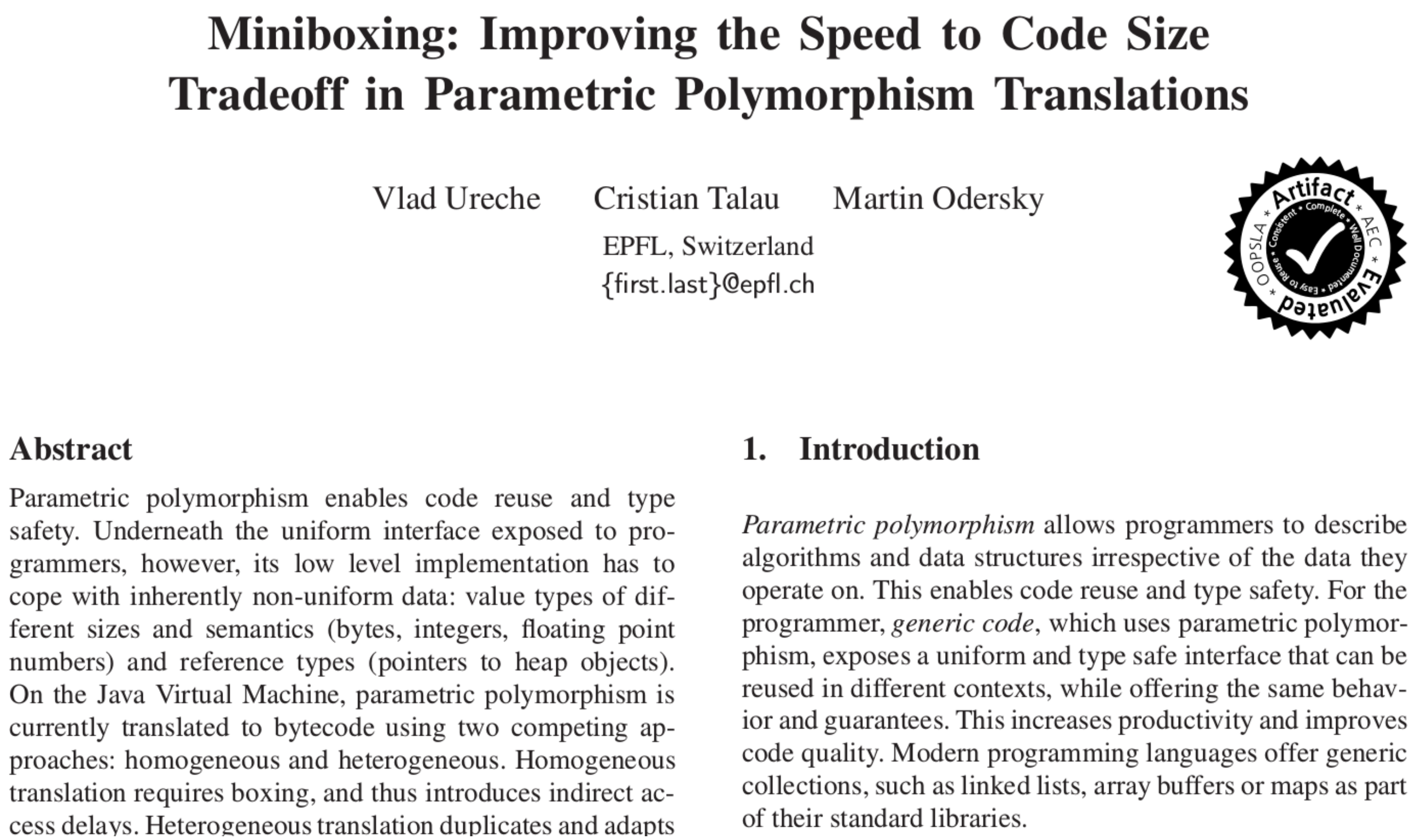
Representation Transformations

- Unboxing Primitive Types
- Value Classes
- Miniboxing



Miniboxing

OOPSLA '13



Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations

Vlad Ureche Cristian Talau Martin Odersky
EPFL, Switzerland
{first.last}@epfl.ch

Abstract

Parametric polymorphism enables code reuse and type safety. Underneath the uniform interface exposed to programmers, however, its low level implementation has to cope with inherently non-uniform data: value types of different sizes and semantics (bytes, integers, floating point numbers) and reference types (pointers to heap objects). On the Java Virtual Machine, parametric polymorphism is currently translated to bytecode using two competing approaches: homogeneous and heterogeneous. Homogeneous translation requires boxing, and thus introduces indirect access delays. Heterogeneous translation duplicates and adapts

1. Introduction

Parametric polymorphism allows programmers to describe algorithms and data structures irrespective of the data they operate on. This enables code reuse and type safety. For the programmer, *generic code*, which uses parametric polymorphism, exposes a uniform and type safe interface that can be reused in different contexts, while offering the same behavior and guarantees. This increases productivity and improves code quality. Modern programming languages offer generic collections, such as linked lists, array buffers or maps as part of their standard libraries.

scala-ldl.org

Miniboxing

T **(primitive)**

Miniboxing

T (primitive)

long integer

- preferred encoding
- for all primitive types

Miniboxing

T (primitive)

☺ **long integer**

- preferred encoding
- for all primitive types

T (erased to Object) ☹

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

Miniboxing

T (primitive)

$$\overbrace{\hspace{10em}}$$

☺ **long integer**

- preferred encoding
- for all primitive types

T (erased to Object) ☹

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

scalac { ① Choice of representation / ② Coercions between representations

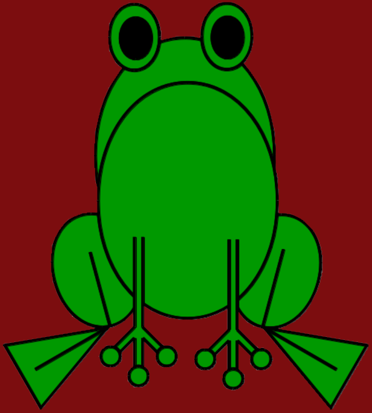
Representation Transformations

- Unboxing Primitive Types
- Value Classes
- Miniboxing

Representation Transformations

- Unboxing Primitive Types
- Value Classes
- Miniboxing

$\Bigg\}$ motivated by erased generics



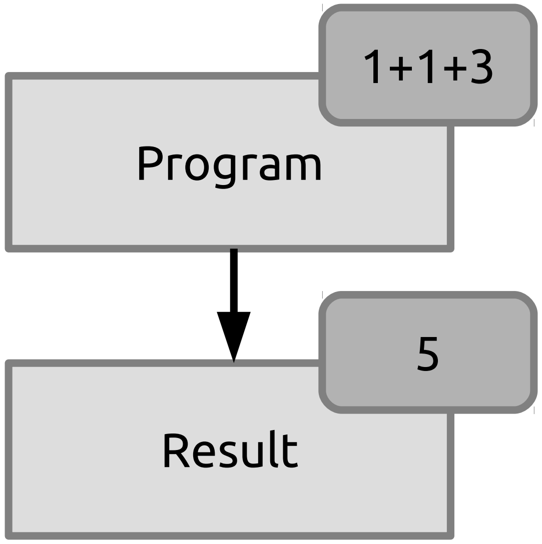
Representation Transformations

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

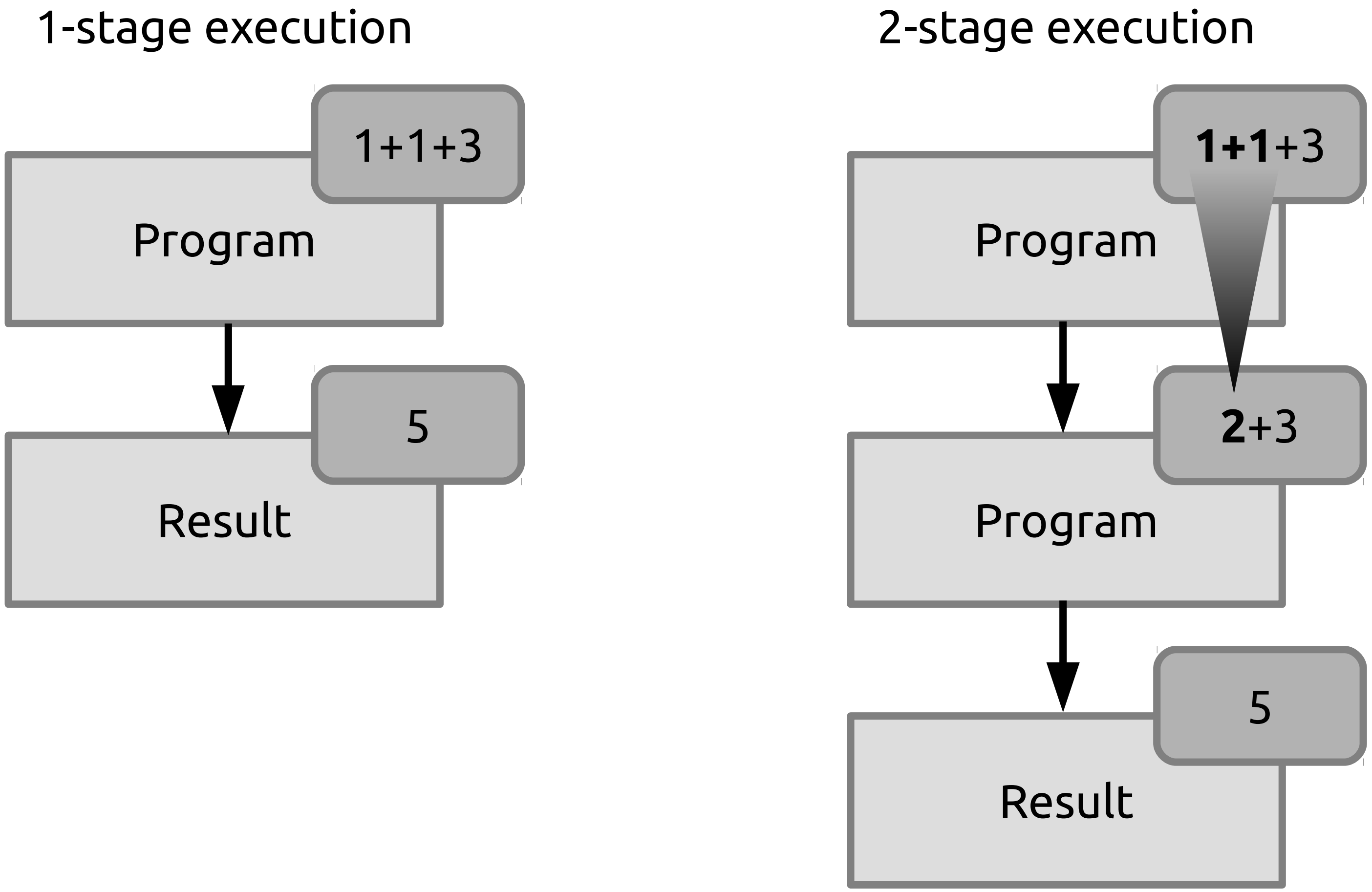
$\left.\begin{array}{l}\text{Unboxing Primitive Types}\\ \text{Value Classes}\\ \text{Miniboxing}\end{array}\right\}$ motivated by erased generics

Staging

1-stage execution



Staging



1-stage execution
1+1+3
Program
5
Result
2-stage execution
1+1+3
Program
2+3
Program
5
Result

Staging

$$\overbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{\textbf{value}}$$

Staging

value

direct value (5)

- is **a computed value**
- from an expression evaluated in the current stage

Staging

value

direct value (5)

- is **a computed value**
- from an expression evaluated in the current stage

lifted expression (2+3)

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

Staging

$$\overbrace{\text{direct value (5)} \qquad\qquad \text{lifted expression (2+3)}}^{\textbf{value}}$$

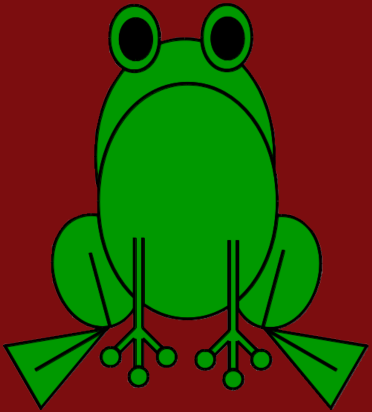
direct value (5)

- is **a computed value**
- from an expression evaluated in the current stage

lifted expression (2+3)

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

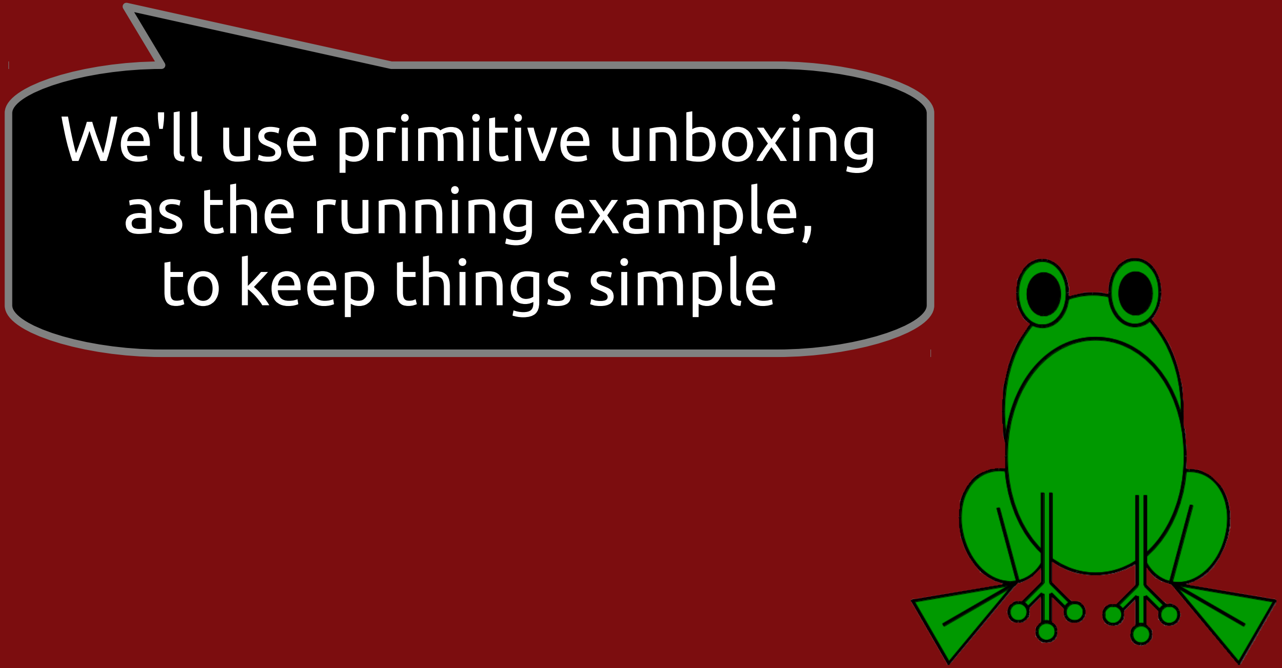
scalac {
① ~~Choice of representation~~ – domain-specific
② Coercions between representations



scala-ldl.org

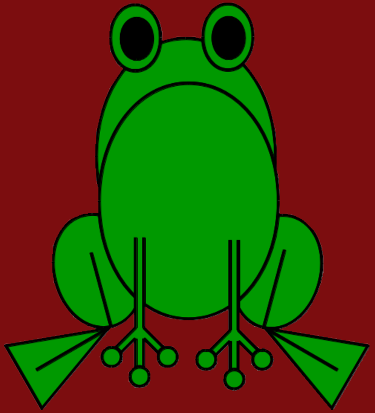
How to transform a program?

How to transform a program?



○ Syntax-based transformation

○ Late Data Layout transformation



● Syntax-based transformation

○ Late Data Layout transformation

Syntax-based

- we need **coercions** between representations
- simple set of syntax-based rules
  - example

Syntax-based

```
val x: Int = ...
val y: Int = x
```

Syntax-based

```
val x: Int = ...
val y: Int = x
```

Syntax-based

```
val x: Int = ...
val y: Int = x
```

```
val x: int = unbox(...)
val y: Int = box(x)
```

Syntax-based

```
val x: Int = ...
val y: Int = x
```

⇩

```
val x: int = unbox(...)
val y: Int = box(x)
```

Coerce the definition right-hand side

Syntax-based

```
val x: Int = ...
val y: Int = x
```

⇩

```
val x: int = unbox(...)
val y: Int = box(x)
```

Coerce all occurences of the transformed value

Syntax-based

```
val x: Int = ...
val y: Int = x
```

⇩

```
val x: int = unbox(...)
val y: Int = box(x)
```

Syntax-based

```
val x: Int = ...
val y: Int = x
```

$\Downarrow$

```
val x: int = unbox(...)
val y: Int = box(x)
```

$\Downarrow$

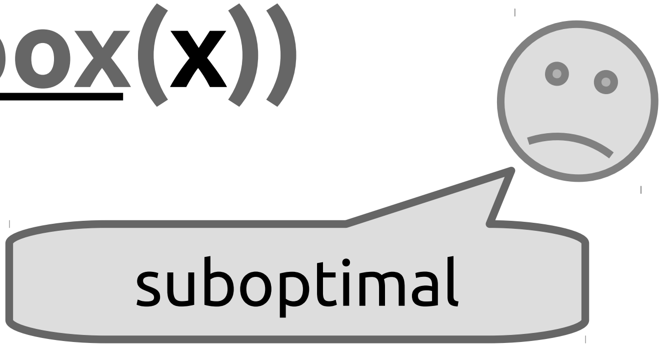
```
val x: int = unbox(...)
val y: int = unbox(box(x))
```

Syntax-based

```
val x: Int = ...
val y: Int = x
```

```
val x: int = unbox(...)
val y: Int = box(x)
```

```
val x: int = unbox(...)
val y: int = unbox(box(x))
```



Peephole Optimization

$$\begin{tikzcd}
\texttt{val y: int = unbox(box(x))} \arrow[d, "\text{peephole}"] \\
\texttt{val y: int = x}
\end{tikzcd}$$

Syntax-based

another example

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

Yes, unboxing the returned value

```
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 =
  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)
```

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 =
  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)
 )
```

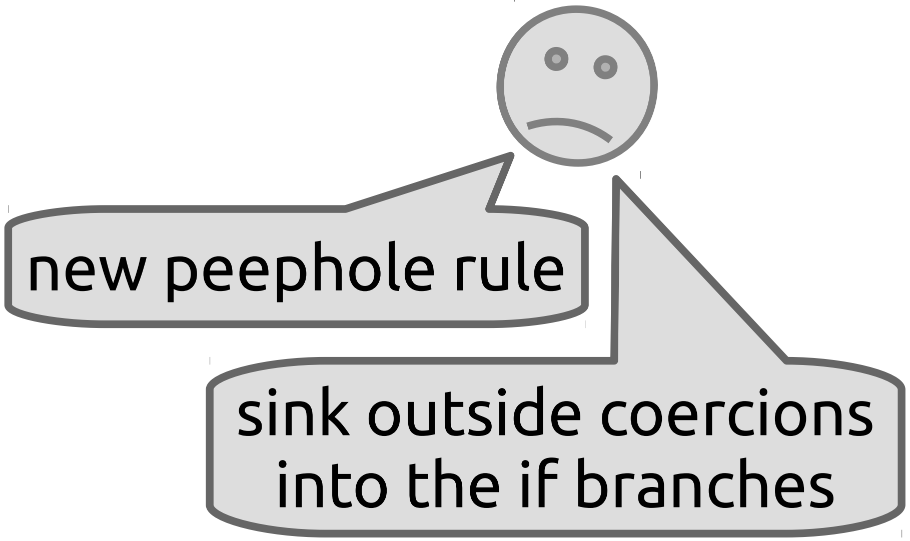
Syntax-based

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

new peephole rule

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

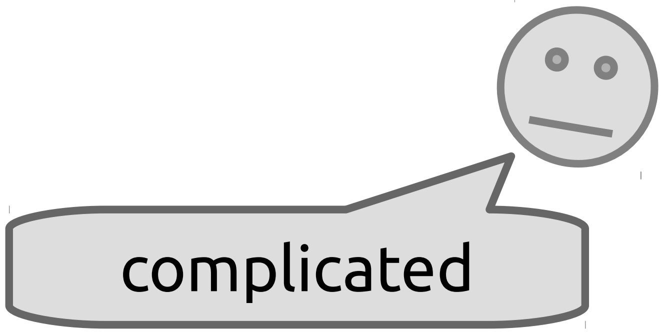
```
def choice(t1: int, t2: int): int =
  if (Random.nextBoolean())
    unbox(box(t1))
  else
    unbox(box(t2))
```

Syntax-based

```
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 <u>**multiple**</u> **rewrite rules** for each node
  - needs <u>**stateful**</u> **rewrite rules**
  - leads to an explosion of rules **x** states

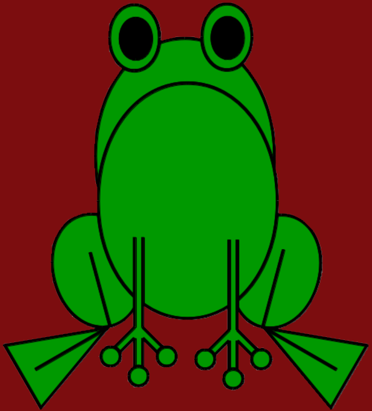
Details in
the paper

Coercions are **fixed in the tree**

Coercions are **fixed in the tree** and moving them around is difficult.

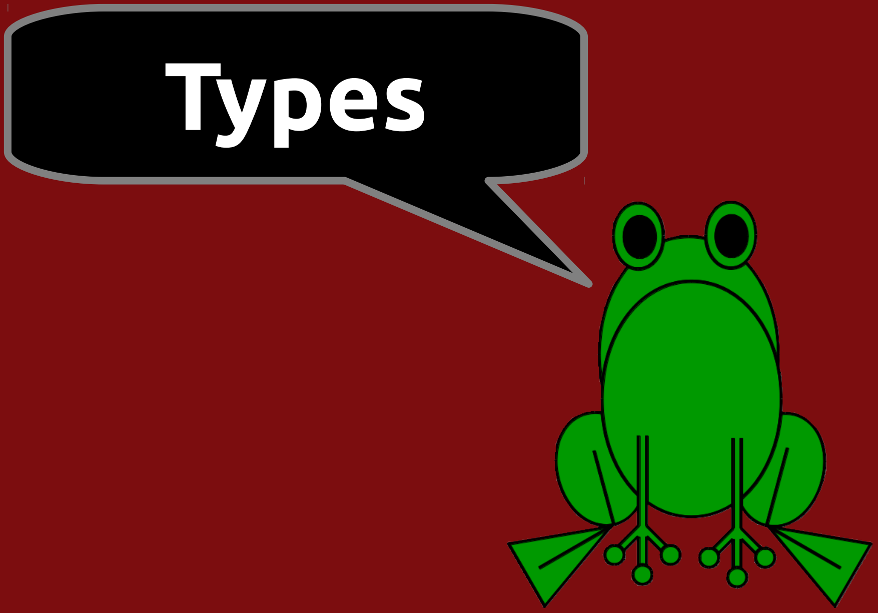
Coercions are **fixed in the tree** and moving them around is difficult.

We need a more **fluid abstraction**.



Coercions are **fixed in the tree** and moving them around is difficult.

We need a more **fluid abstraction**.



○ Syntax-based transformation

● Late Data Layout transformation

Late Data Layout transformation

Phases

- ● **Inject**
- ○ **Coerce**
- ○ **Commit**

LDL Transformation
The Inject Phase

- propagates representation information
  - into the type system
    - based on annotated types
    - e.g. an **@unboxed** annotation added to integers

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
```

LDL Transformation

● inject
○ coerce
○ commit

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

depending on the transformation, other operations can be performed as well (e.g. miniboxing duplicates methods)

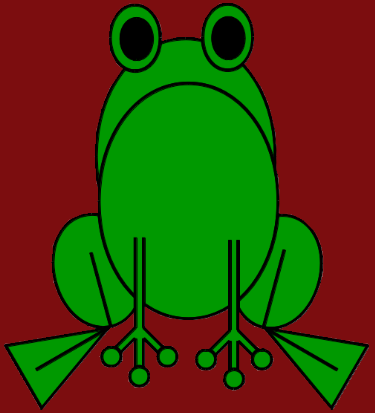
○ Late Data Layout transformation

Phases

○ **Inject**

● **Coerce**

○ **Commit**



LDL Transformation
The Coerce Phase

- introduces coercions
  - re-type-checks the tree
  - exposes representation mismatches
    - as **annotation** mismatches (**Int** vs **@unboxed Int**)
    - leading to coercions

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
```

scala-ldl.org

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
```

$\boxed{\texttt{t1}}$

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:

...

scala-ldl.org

LDL Transformation

inject

coerce

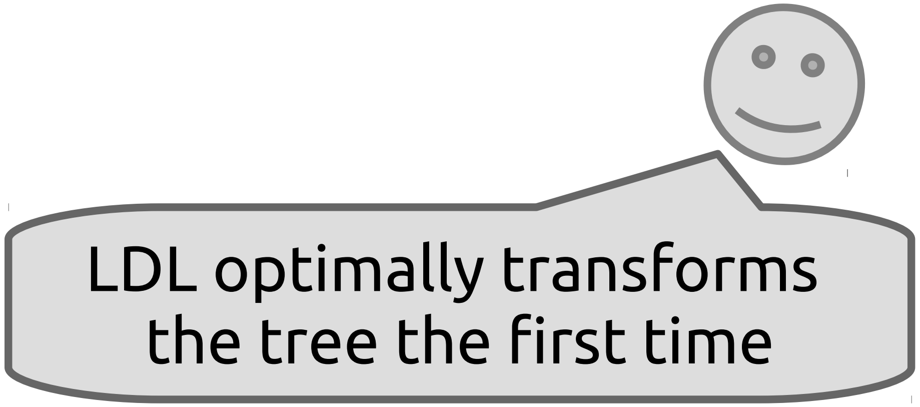
commit

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

LDL Transformation

- inject
- **coerce**
- commit

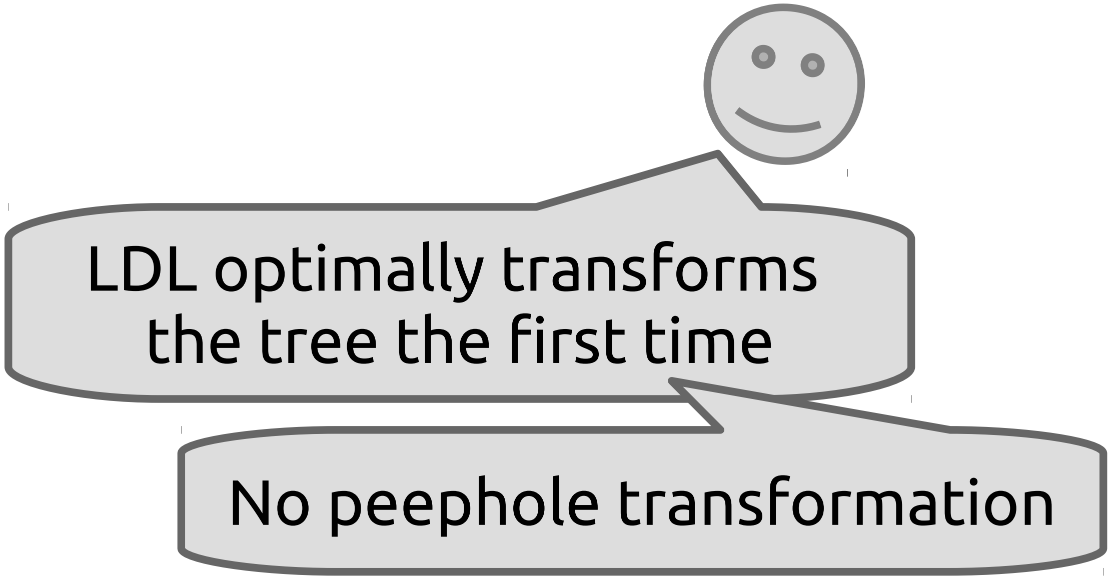
```
def choice(t1: @unboxed Int,
           t2: @unboxed Int): @unboxed Int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



LDL Transformation

○ inject
● **coerce**
○ commit

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


LDL optimally transforms the tree the first time
No peephole transformation

○ Late Data Layout transformation

Phases

- ○ **Inject**
- ○ **Coerce**
- ● **Commit**

scala-ldl.org

LDL Transformation

The Commit Phase

- converts annotations to representations
  - **@unboxed Int → int**
  - **Int → java.lang.Integer**
- coercion markers are also transformed
  - **box(...) → new Integer(...)**
  - **unbox(...) → ....intValue**

LDL Transformation

○ inject
○ coerce
● **commit**

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

LDL Transformation

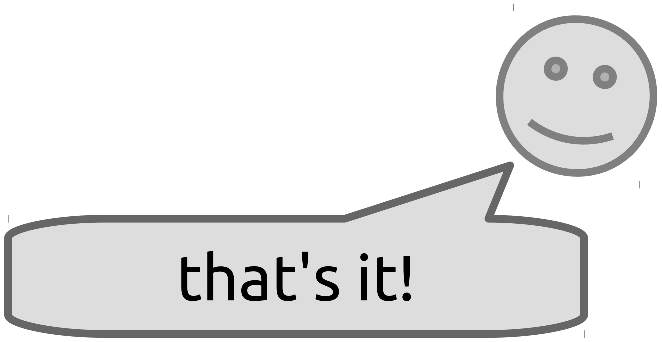
○ inject
○ coerce
● **commit**

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

LDL Transformation

inject
coerce
commit

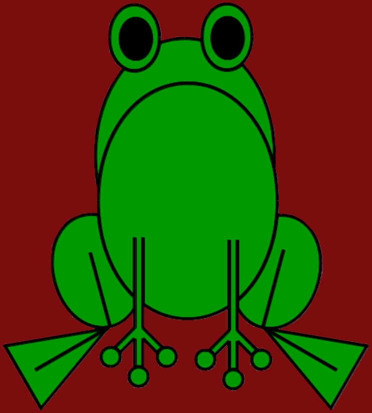
```
def choice(t1: int,
           t2: int): int =
  if (Random.nextBoolean())
    t1
  else
    t2
```



Late Data Layout transformation

Phases

- **Inject**
- **Coerce**
- **Commit**



Motivation

Transformation

$\boxed{\text{Properties}}$

Benchmarks

Conclusion

scala-ldl.org

○ Late Data Layout transformation

Properties	●	Selectivity
	○	Consistency
	○	Optimality (not formally proven yet)

Selectivity

● **inject**
○ coerce
○ commit

- annotated types
  - **selectively** pick the representation for each value

Selectivity

● **inject**
○ coerce
○ commit

- annotated types
  - **selectively** pick the representation for each value
- selectivity is used for
  - bridge methods (some args boxed, others unboxed)
  - value classes (JVM: no multi-value returns)
  - staging (representation: domain-specific knowledge)
    - **List[Int]** vs **List[@staged Int]** vs **@staged List[Int]**

Selectivity

● **inject**
○ coerce
○ commit

```
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

○ inject
● **coerce**
○ commit

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

Selectivity

○ inject
○ coerce
● **commit**

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

Selectivity

inject

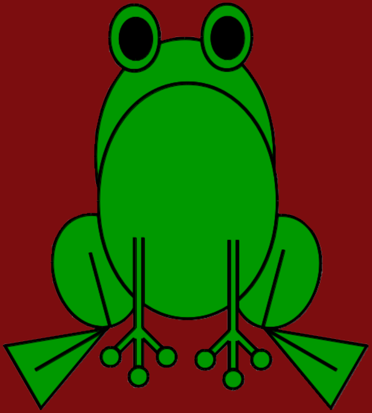
coerce

commit

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

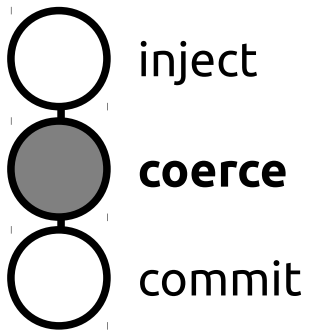
○ Late Data Layout transformation

Properties
- ○ Selectivity
- ● Consistency
- ○ Optimality (not formally proven yet)



scala-ldl.org

Consistency



- representations become **part of types**
- re-type-checking the program
  - proves type correctness
  - proves **representation consistency**

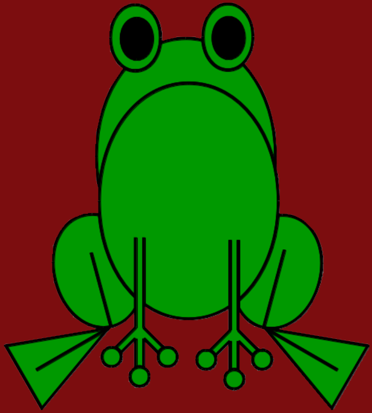
○ Late Data Layout transformation

Properties

○ Selectivity

○ Consistency

● Optimality (not formally proven yet)



scala-ldl.org

Optimality

○ inject
○ coerce
● **commit**

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

Optimality

○ inject
○ coerce
● **commit**

execution

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

Optimality

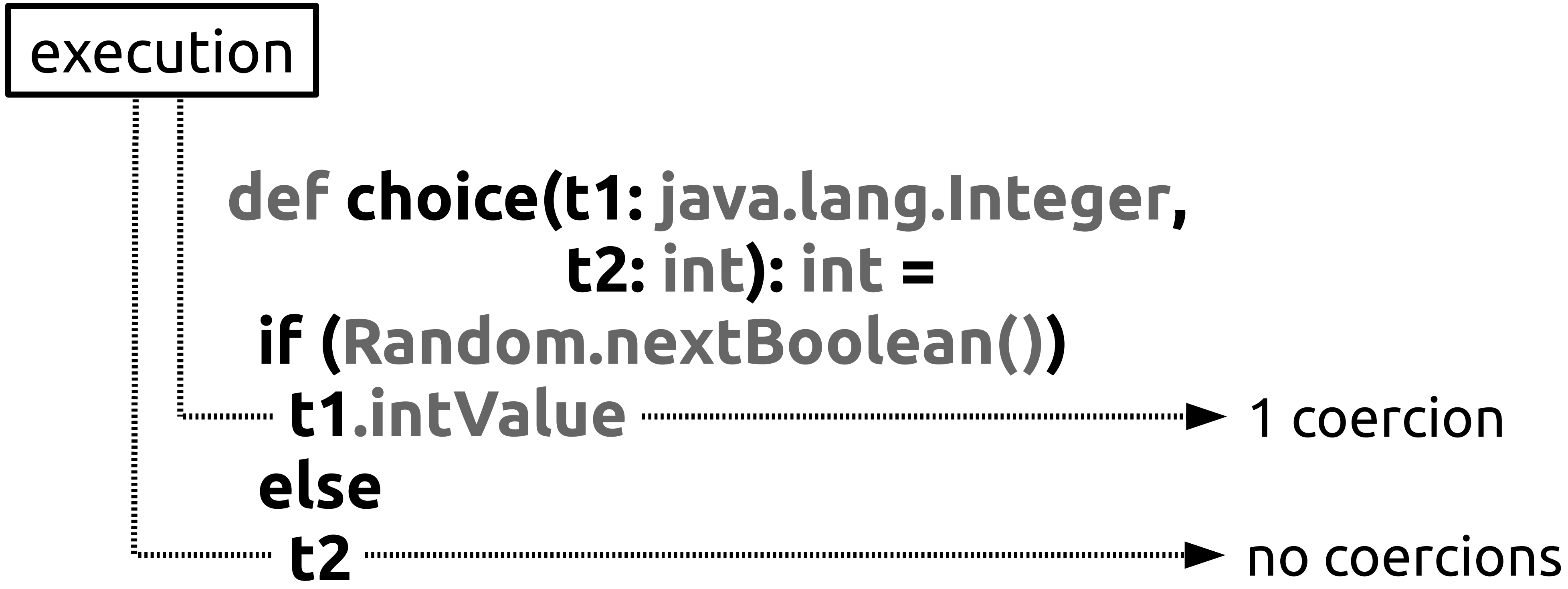
○ inject
○ coerce
● **commit**

execution

```
def choice(t1: java.lang.Integer,
                t2: int): int =
  if (Random.nextBoolean())
    t1.intValue                    ► 1 coercion
  else
    t2
```

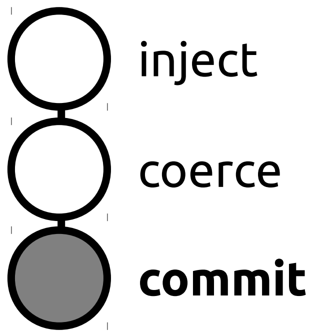
Optimality

○ inject
○ coerce
● **commit**



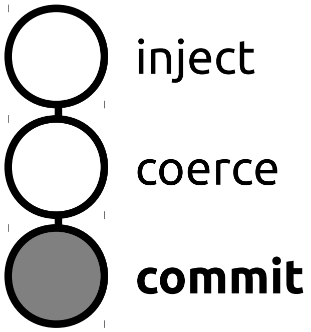
```
def choice(t1: java.lang.Integer,
           t2: int): int =
  if (Random.nextBoolean())
    t1.intValue
  else
    t2
```

Optimality



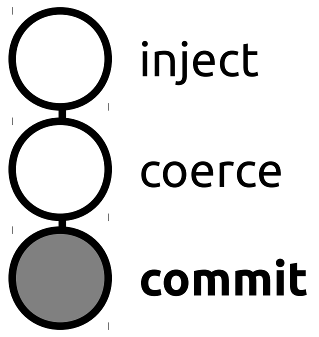
- on any execution trace through the program
  - the number of **coercions executed is minimum**
  - assuming the program terminates

Optimality



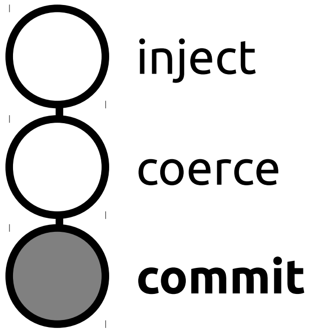
- on any execution trace through the program
  - the number of **coercions executed is minimum**
  - assuming the program terminates
- modulo
  - annotations introduced by the **inject phase**
    - unbox both parameters → **no coercions at all**

Optimality



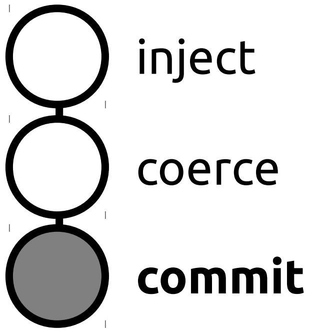
- on any execution trace through the program
  - the number of **coercions executed is minimum**
  - assuming the program terminates
- modulo
  - annotations introduced by the **inject phase**
    - unbox both parameters → **no coercions at all**
  - post-transformations done by the **commit phase**
    - **box(...) → new Integer(new Integer(...).intValue)**

Optimality



- peephole optimization
  - propagates coercions
- type system
  - propagates types

Optimality



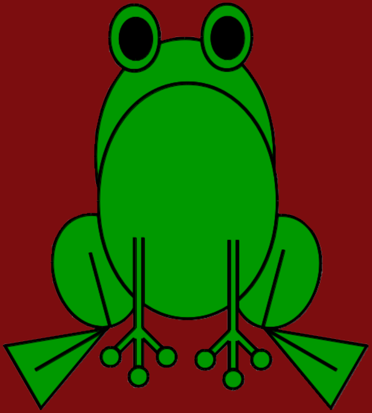
- peephole optimization
  - propagates coercions
- type system
  - propagates types
  - **but types are fluid whereas coercions are not**

details in the paper

Late Data Layout transformation

Properties

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



scala-ldl.org

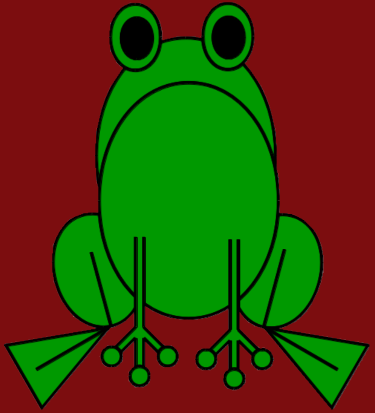
Motivation

Transformation

Properties

$\boxed{\text{Benchmarks}}$

Conclusion



scala-ldl.org

LDL is used in

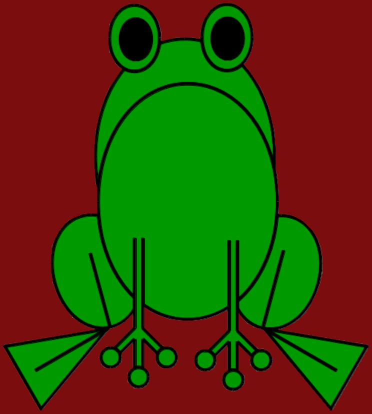
- Scala compiler plugins
  - miniboxing (specialization)
  - value-class plugin
  - staging plugin

Benchmarks
… in the paper

- implementation effort
  - Late Data Layout mechanism
    - developed as part of miniboxing
    - reused by the other compiler plugins
  - value class plugin → **2 developer-weeks**
  - staging plugin → **1 developer-week**

Benchmarks
… in the paper

- performance
  - baseline vs transformed code
- numbers
  - up to **2x** speedup when transforming **value classes**
  - up to **22x** speedup when using **miniboxing**
  - up to **59x** speedup when **staging**



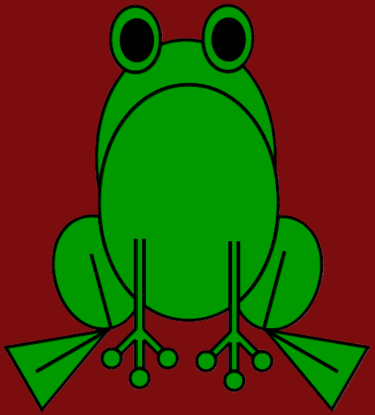
Conclusion
Insights

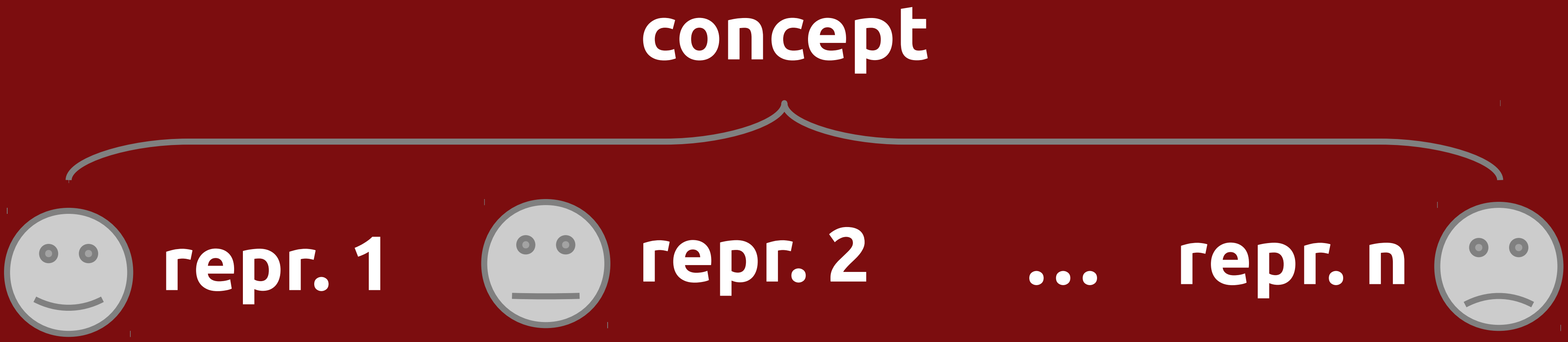
- use **annotated types**
  - to **selectively** mark values with the representation
- use **expected type** propagation
  - to provide **optimal** transformation
- use the **type system**
  - to provide representation **consistency**

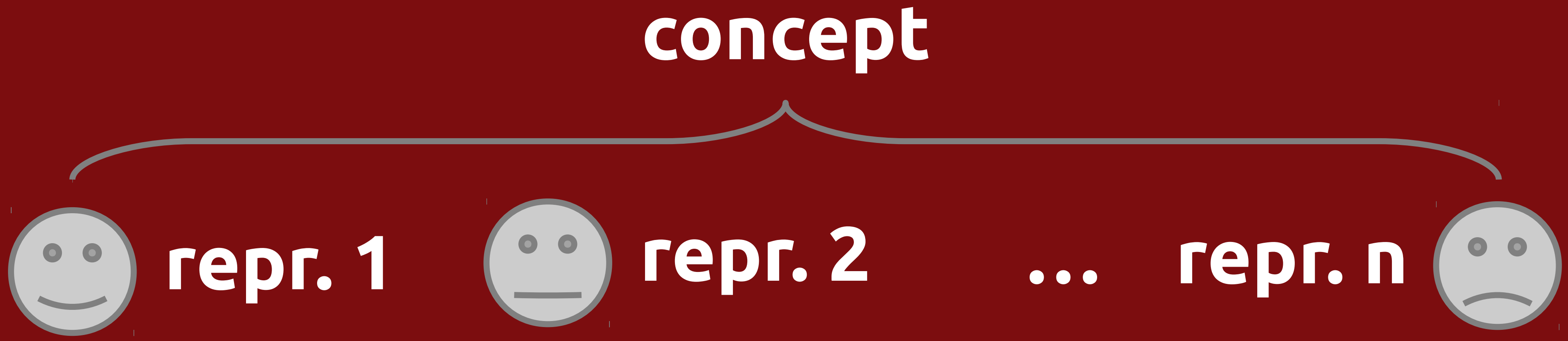
Credits and Thank you-s

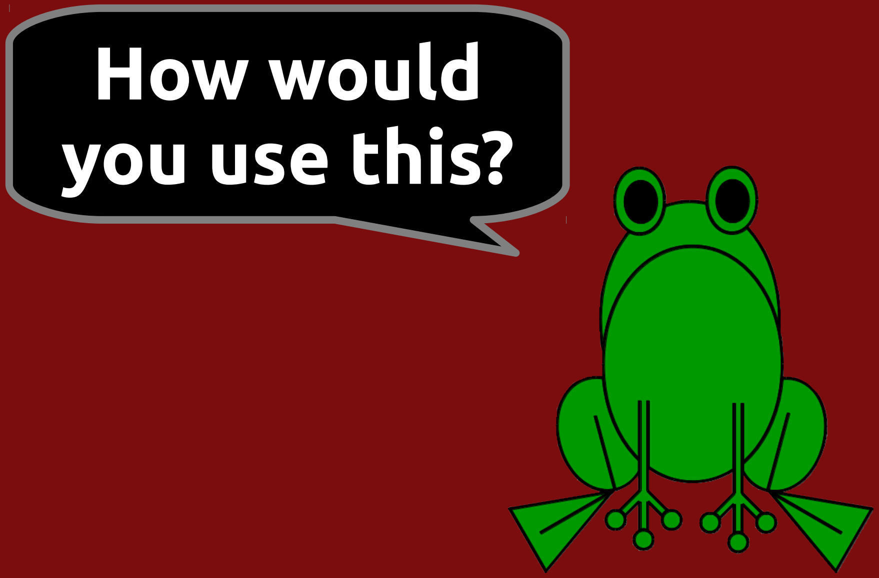
- Cristian Talau - developed the initial prototype, as a semester project
- Eugene Burmako - the value class plugin based on the LDL transformation
- Aymeric Genet - developing collection-like benchmarks for the miniboxing plugin
- Martin Odersky, for his patient guidance
- Eugene Burmako, for trusting 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
- Xiaoya Xiang and Philip Stutz for trusting miniboxing enough to try it out

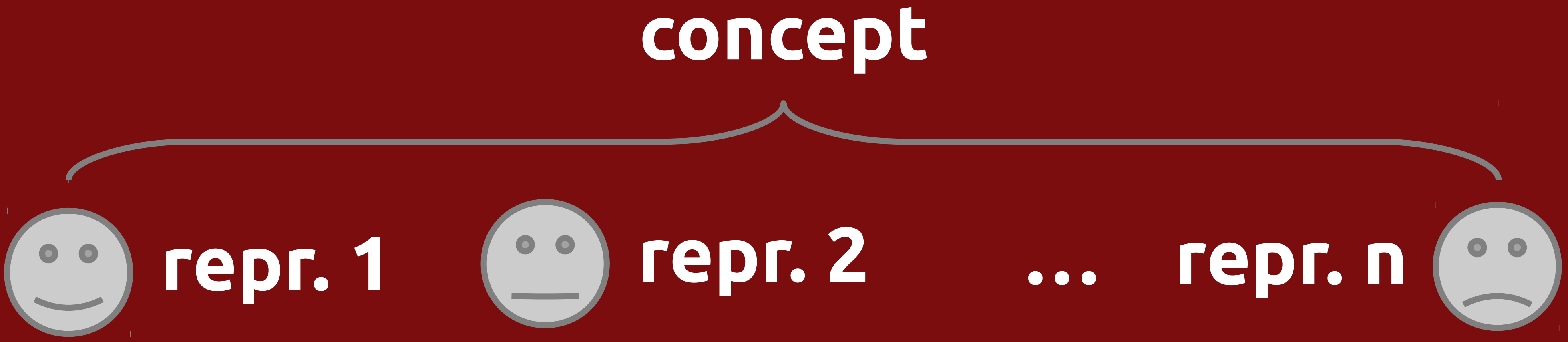
Special thanks to the Scala Community for their support!
(@StuHood, @vpatryshev and everyone else!)




concept
repr. 1
repr. 2
...
repr. n


concept
repr. 1
repr. 2
...
repr. n


How would you use this?



scala-ldl.org