Late Data Layout: Unifying Data Representation Transformations

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

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

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 transformations. In doing so, we make two key observations: (1) annotated types conveniently capture the semantics of using multiple representations and (2) local type inference can be used to consistently and optimally introduce coercions.

We validated our approach by implementing three language features as Scala compiler extensions: value classes, specialization (using the miniboxing representation) and a simplified multi-stage programming mechanism.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Polymorphism; E.2 [*Object representation*]

Keywords Data Representation; Object-oriented; Annotated Types; Type Systems; Local Type Inference

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 (unboxed) representation. Thus, in the low-level compiled code, `x` is using the unboxed representation, denoted as `int`:

```
def identity(arg: Object): Object = arg
// val x: Int = identity[Int](5):
val arg_boxed: Object = box(5)
val ret_boxed: Object = identity(arg_boxed)
val x: int = unbox(ret_boxed)
```

The low-level code shows the two representations of the high-level `Int` concept: the unboxed primitive `int` and the boxed `Object`, which is compatible with erased generics. There are two approaches to exposing this duality in programming languages: In Java, both representations are accessible to programmers, making them responsible for the choice and exposing the language feature interactions. On the other hand, in order to avoid burdening programmers with implementation details, languages such as ML, Haskell and Scala expose a unified concept, regardless of its representation. Then, during compilation, the representation is automatically chosen based on the interaction with the other language features and the necessary coercions between representations, such as `box` and `unbox`, are added to the code.

This strategy of exposing a unified high-level concept with multiple representations is used in other language features as well:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

OOPSLA '14, October 19-21 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2585-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2660193.2660197

Value classes [2, 7, 29] behave as classes in the object-oriented hierarchy, but are optimized to efficient C-like structures [66] where possible. This exposes two representations of the value class concept: an inline, efficient struct representation and a flexible object-oriented representation that supports subtyping and virtual method calls.

Specialization [21, 22, 28] is an optimized translation for generics, which compiles methods and classes to multiple variants, each adapted for a primitive type. An improvement to specialization is using the miniboxed representation [5, 75], which creates a single variant for all primitive types, called a minibox. In this transformation, a generic type `T` can be either boxed or miniboxed, in yet another instance of a concept with multiple representations.

Multi-stage programming (also referred to as "staging") [70] allows executing a program in multiple stages, at each execution stage generating a new program that is compiled and run, until the final program outputs the result. In practice, this technique is used to lift expressions to operation graphs and to generate new, optimized code for them. This shows a very different case of dual representations: a value can be represented either as itself or as a lifted expression, to be evaluated in a future execution stage.

The examples above seem like unrelated language features. And, indeed, compiler implementers have provided dedicated solutions for each of them, entangling the core transformation mechanism with assumptions about the language and platform. For instance, the solutions employed by ML and Scala are aimed at satisfying the constraints of erased generics [15, 42, 72], and hardcode this decision into their rewriting algorithm. Miniboxing uses a custom transformation implemented as a Scala compiler plugin [5, 75], aimed at only the miniboxed representation. Finally, the Lightweight Modular Staging framework [55] in Scala relies on a custom fork of the main compiler, dubbed Scala-Virtualized [45], which is specifically retrofitted to support lifting language constructs.

Yet, these transformations share two common traits: (1) the use of multiple representations for the same concept and (2) the automatic introduction of coercions between these representations during program compilation. These similarities suggest there is an underlying principle that generalizes the individual algorithms. We believe exposing this principle can disentangle the transformations from their assumptions, providing a framework that researchers can formally reason about and that implementors can reuse when developing new transformations.

To this end, we present an elegant and minimalistic type-driven mechanism that uses annotations to guide the introduction of coercions between alternative representations, which we call the Late Data Layout (LDL) mechanism. In doing so, we make the following contributions:

- We survey the existing approaches to data representation transformation, show their limitations and explore the additional features required (§2 and §3);
- We show the Late Data Layout (LDL) representation transformation mechanism, which does not impose the semantics of alternative representations and coercions (§4) and reason intuitively about its properties (§5);
- We validate the mechanism by implementing three language features as Scala compiler extensions using the LDL transformation: value classes[1], specialization using the miniboxing representation[2] and a simple staging mechanism[3] (§6). For each of these use cases, we describe the implementation in detail, we compare it to the existing transformations in terms of code size and complexity, we evaluate the resulting programs in terms of performance and finally show the specific extensions we added to the LDL mechanism to support each use case.

The Late Data Layout mechanism relies on two key insights: (1) annotated types conveniently capture the semantics of using multiple representations and (2) local type inference can be used to consistently and optimally introduce coercions between these representations. The following paragraphs describe the insights and how they influence the mechanism.

Key Insights

Through annotations, additional metadata can be attached to the types in a program [4, 8]. This, in turn, allows external plugins to verify more properties of the code while leveraging the existing type system infrastructure. Annotated types have been used to statically check a wide range of program properties, from simple non-`null`-ness to effect tracking and purity analysis [51, 60].

Our first key insight is that annotated types are a perfect match for encoding the multiple representations of a high-level concept. For example, changing a value's type from `Int` to `@unboxed Int` marks it for later unboxing. This provides *generality*, *selectivity* and *automation*.

Generality. The annotations can be introduced either automatically, by the compiler, based on the interactions of different language features, or manually, by programmers. This provides the flexibility necessary to capture a wide variety of transformations: some of them work automatically, like unboxing primitive types and value classes, whereas others, like staging, require manual annotation, corresponding to domain-specific knowledge.

Selectivity. Annotated types allow selectively marking values with their alternative representation. For example, marking a value's type as `@unboxed` means it will use the alternative unboxed representation. Contrarily, leaving it unmarked will continue to use the default, boxed, representation. In the following example, we show how simple it is for the compiler to signal whether a value should be boxed or unboxed and whether generics are erased, as in Java, or specialized, as in the .NET CLR [9, 35]:

[1] http://github.com/miniboxing/value-plugin

[2] http://github.com/miniboxing/miniboxing-plugin

[3] http://github.com/miniboxing/stage-plugin

```
// erased generics, boxed value:
val x: Int = identity[Int](5)
// erased generics, unboxed value:
val x: @unboxed Int = identity[Int](5)
// specialized generics, unboxed value:
val x: @unboxed Int = identity[@unboxed Int](5)
```

This flexibility of annotating individual values with their alternative representation is in sharp contrast to state of the art data representation transformations [15, 42]. These transformations consider the unboxed representation as always desirable and hardcode the semantics of erased generics into their transformation rules. Section §3.3 shows that being able to selectively annotate the values that use a different representation is crucial to implementing transformations in object-oriented languages. This flexibility is also fundamental to multi-stage programming, where the choice of execution stage has to be done for each individual value.

Automation. The semantics of annotated types can be specified externally and can change as the compilation pipeline advances: keeping annotated and non-annotated types compatible emulates the unified concept, allowing seamless inter-operation regardless of the representation. Later, making annotated types incompatible emulates the difference between representations, automatically triggering the introduction of coercions.

Our second key insight is that local type inference [48, 52] can be used to *consistently* and *optimally* introduce coercions based on the annotated types. Once the unified concept has been refined into several representations by making annotated types incompatible, type-checking the program's abstract syntax tree (AST) again reveals the representation inconsistencies, where coercions are required.

Optimality. Name resolution and type propagation can be seen as a forward data flow analysis [36] that, through annotated types, propagates the data representation. On the other hand, local type inference [48, 52] propagates expected types from the outer expressions, providing a backward data flow analysis. Having these two analyses meet at points where the representation doesn't match ensures that coercions are introduced only when necessary, in an optimal way:

```
// erased generics, boxed value:
val x: Int = identity(box(5))
// erased generics, unboxed value:
val x: @unboxed Int = unbox(identity(box(5)))
// specialized generics, unboxed value:
val x: @unboxed Int = identity[@unboxed Int](5)
```

Consistency. Type checking a program means proving its correctness with respect to the theory introduced by the types. Therefore, making representation information available to the type system allows it to prove correctness with respect to the representations in use and the coercions introduced between them, thus proving consistency.

Generality again. The last step of the transformation gives the annotated types their final semantics, by making the alternative representations explicit. For example, primitive unboxing replaces `@unboxed Int` by `int` and gives the coercions, `box` and `unbox`, their semantics: in this case creating the boxed object and reading the unboxed integer from the object representation. This allows the rest of the transformation to work regardless of the actual alternative representations, thus isolating the general mechanism from the representation semantics.

Being type-driven, our approach can be seen as a generalization of the work of *Leroy* on unboxing primitive types in ML [42]. Yet, it is far from a trivial generalization: (1) we introduce the notion of selectively picking the representation for each value, which is crucial to enabling staging, specialization and creating bridge methods [19], (2) we extend the transformation to work in the context of object-oriented languages, with the complexities introduced by subtyping and virtual method calls and (3) we disentangle the transformation from the assumptions that generics are erased and that the alternative representation is always desirable.

In the following sections we explain the motivation for the Late Data Layout mechanism, present it in detail and validate our approach.

2. Data Representation Transformations

In this section we present several approaches to transforming the data representation, highlighting their strong and weak points on small examples. We start with a naive approach, continue with a syntax-based transformation that eagerly introduces coercions and conclude with a type-driven transformation, which only introduces coercions when necessary. To facilitate the presentation, the examples refer to unboxing primitive types, but the explanations can be generalized to all the three use cases described in the validation section: value classes, miniboxing and staging.

In the rest of the paper we consider the integer concept to be boxed by default and represent it by `Int`. The goal of the transformations is to use the unboxed integer, `int`, whenever possible. Unless otherwise specified, all generic classes are assumed to be compiled to erased homogeneous low-level code. Finally, to improve readability, we place annotations in front types (e.g. `@unboxed Int`) instead of after (e.g. `Int @unboxed`), as the Scala syntax requires.

2.1 Naive Transformations

To begin, let us analyze a simple code snippet, where we take the first element of a linked list of integers (`List[Int]`) and construct a new linked list with this one element:

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

A naive approach to compiling down this code would be to replace all boxed integers by their unboxed representations without performing any data-flow analysis:

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

The resulting code is invalid. In the first statement, `x` is unboxed while the right-hand side of its definition, the head of the generic list, is boxed. In the second statement, we create a generic list, which expects the elements to be boxed. Yet, `x` is now unboxed. This example motivates a more elaborate transformation for unboxing integers.

2.2 Eager (Syntax-driven) Transformations

The previous example shows that naively replacing the representation of a value is not enough: we need to patch the definition site and all the use sites, coercing to the right representation:

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

In the snippet above, two coercions have been introduced. In the first line, since `x` becomes unboxed, the right-hand side of its definition also needs to be unboxed. In the second line, `x` is boxed to satisfy the list constructor. This means that by eagerly adding coercions we can keep the program code consistent. Let us take another example:

```
val a: Int = ...
val b: Int = a
```

Since `a` is transformed from boxed to unboxed, all its occurrences are replaced by `box(a)`:

```
val a: int = unbox(...)
val b: Int = box(a)
```

When `b` is transformed, its right hand side is unboxed:

```
val a: int = unbox(...)
val b: int = unbox(box(a))
```

The definition of `b` is suboptimal: it boxes `a` just to unbox it immediately after. In some cases, thanks to escape analysis [65], the Java Virtual Machine just-in-time compiler [38, 50] can remove redundant boxing and unboxing operations. Yet it typically takes 10000 executions to trigger the optimizing just-in-time compiler [39], which means 10000 boxed integers are created just to be immediately unboxed and garbage collected later. And escape analysis is a best-effort optimization, as there are no guarantees on the patterns it will optimize. It would therefore be best if the data representation transformation would eliminate redundant coercions from the start. This is where the peephole optimization comes in.

2.3 Peephole Optimization For Eager Transformations

A peephole optimization [32, 75] can be used to remove the redundant coercions introduced by an eager (syntax-driven) transformation. The name “peephole” comes from the very limited scope of the rewriting rules, usually encompassing a coercion and another abstract syntax tree node. For example, the peephole optimization rewrites `box(unbox(t))` and `unbox(box(t))` to just `t`. This simple rewrite rule eliminates the redundant coercions in the definition of `b`. Yet, it is not enough.

Unboxed operations. Let us take an example operation between two boxed values, where `a` and `b` are the values defined in the previous section:

```
val c: Int = a + b
```

Eager transformations box `a` and `b` and unbox the result of their addition, which is inefficient:

```
val c: int = unbox(box(a) + box(b))
```

Therefore, we need an extra rule for distributing the unboxing operation inside: `unbox(t1 + t2)` ⇒ `unbox(t1) _+_ unbox(t2)`, where `_+_` is the platform-provided intrinsic unboxed integer addition. With this extra rule, coupled with coercion elimination, the expression is fully optimized.

Conditional optimization. The previous rule is not enough to produce optimal code in all cases:

```
def foo(x: Int, y: Int): Int =
  if (...) x else y
```

In order to optimize the `foo` method, the compiler unboxes `x`, `y` and the return type of `foo` and introduces three coercions: two for boxing `x` and `y` back and one for unboxing the body of `foo`:

```
def foo(x: int, y: int): int =
  unbox(if (...) box(x) else box(y))
```

In this case, we need a rule for distributing the coercion surrounding an `if` node to its branches: `unbox(if (...) a else b)` → `if (...) unbox(a) else unbox(b)`:

```
def foo(x: int, y: int): int =
  if (...) unbox(box(x)) else unbox(box(y))
```

Which in turn is completely optimized by the first rule, `unbox(box(t))` → `t`.

Block optimizations. Let us take one final example:

```
def bar(): Boolean = {
  foo(..., ...)
  true
}
```

Since the type of `foo` was transformed, any call to it needs to be adapted: integer arguments need to be unboxed and the result needs to be boxed back:

```
def bar(): Boolean = {
  box(foo(unbox(...), unbox(...)))
  true
}
```

In a block with n expressions, the first $n - 1$ expressions are treated as statements, so their results are ignored. Therefore boxing the result of `foo` is redundant, since the boxed value will be ignored anyway. Thus we have to introduce a specific rule for blocks which removes coercions on statements. Not only that this rule is already stateful, depending on the position in the block, but it is even not sufficient: the last expression in a block, which acts as the block’s result, has the distribution property of `if` conditionals. Furthermore, given multiple stateful rules, they can be mixed together: What if a conditional is nested in a block, in statement position? Should coercions be distributed or ignored?

In practice, a peephole optimization needs multiple stateful rewrite rules for each type of node in the intermediate representation of the program, usually an abstract syntax tree (AST) in the compiler. This suggests that although eager transformations work well for minimalistic intermediate representations, such as Haskell's Core, the number and complexity of AST nodes in the Scala compiler makes a peephole transformation impractical. The initial implementation of miniboxing [75] used an eager transformation but the tedium of maintaining and tweaking the peephole optimization rules led to the development of the Late Data Layout mechanism, which, itself, is based on a type-driven transformation.

2.4 Type-driven Transformations

Syntax-driven transformations are straightforward, but they eagerly introduce coercions, which need to be optimized later. An alternative would be to introduce coercions only when a representation mismatch occurs, using a dedicated mechanism to check representation consistency.

The dedicated mechanism can be the type checker. Indeed, injecting the representation information in the type checker allows it to automatically and reliably detect mismatches, which can be patched by introducing coercions, in a mechanism similar to the implicit conversions of Scala. This achieves optimality out of the box in the case of `foo` shown before, as the type checker knows all variables are unboxed, hence no coercions are necessary:

```
def foo(x: int, y: int): int =
  if (...) x else y
```

This type-driven transformation is a precursor to the Late Data Layout mechanism. Yet, in the current from, type-driven transformations are still not always optimal and not applicable in a general setting. To show why, let us assume we introduce a boxed unsigned integer `UInt`, which we unbox to `int`. The operators for the unsigned type are different, but the unboxed representation is exactly the same as for `Int`. In practice, this is the norm: several value classes can have the same parameter types, so their unboxed representations coincide. Furthermore, all staged expressions share the same alternative representation. Let us consider the following example using the signed `Int` and the unsigned `UInt`:

```
val m: UInt = ...
val n: Int = ...
List[AnyVal](if (...) m else n)
```

Transforming the example, both `m` and `n` are unboxed to `int`, so the `if` expression produces an `int`:

```
val m: int = unbox_uint(...)
val n: int = unbox_int(...)
List[AnyVal](
  if (...) m else n
  // ^ mismatch (expected: AnyVal, found: int)
)
```

The generic linked list constructor expects a boxed argument, but we pass in an unboxed `int`, triggering a mismatch. Thus, the `if` expression needs to be boxed. But what coercion should be used? Should it be `box_uint` or `box_int`? Since the provenance of the expression has been lost, we can't discern between the two. A correct translation would have introduced coercions earlier:

```
val m: int = unbox_uint(...)
val n: int = unbox_int(...)
List[AnyVal](if (.) box_uint(m) else box_int(n))
```

It may seem that transforming values one by one might provide a way out of this conundrum. This way, only a single value at a time would be in flux, which would make it easy to guess the coercion necessary to patch mismatches. However, this takes us back to square one with respect to the suboptimality of the resulting code: transforming one value at a time is equivalent to having an eager transformation, which needs to be consistent at each step and does so by introducing too many coercions. For example, transforming one value at a time would break the first example, the `foo` method, which would end up requiring a peephole optimization:

```
def foo(x: int, y: int): Int = // now to unbox
  if (...) box(x) else box(y) // the foo return
```

Clearly, a different approach is required to make type-driven transformations viable in a general setting. But before going into the Late Data Layout mechanism, we dive into the interaction between object-oriented language features and data representation transformations.

3. Object-Oriented Data Representation

The previous section presented the problems faced by data representation transformations, especially given complex intermediary representations (IRs) such as the one used in the Scala compiler. This section identifies additional challenges introduced by object orientation.

3.1 Subtyping

In object-oriented languages, all reference types have a common super type, usually called `Object`, which provides universal methods such as `toString`, `hashCode` and `equals`. This challenges representation transformations:

```
val a: Int = ...  // can be unboxed
val b: Object = a // needs to be boxed back
```

Although `a` can use the unboxed representation, it needs to be boxed back when it is assigned to `b`, since `b` is compiled to an object reference in the low level code.

This is also the case for value classes: whenever a variable is statically known to hold a value class, it can be optimally represented by its fields. But when the value class is used in a context where a super type is expected, it has to be boxed:

```
trait T
@value class X(val x: Int) extends T
@value class Y(val x: Int) extends T
val x: X = new X(3) // can be unboxed
val y: Y = new Y(31)// can be unboxed
val t: T = if (...) x else y // must be boxed
```

Even though `X` and `Y` unbox to `Int`, unboxing `t` is till not possible, as it would lose the provenance information necessary for boxing: an integer corresponding to the unboxed `t` could have originated from unboxing either `X` or `Y`, but, after unboxing, it would not be known from which. Therefore, to avoid generating incorrect programs, conformance to super types, or up-casting, requires boxing.

3.2 Virtual Method Calls

Virtual method calls also pose challenges for data representation transformations. Boxed objects can act as the receivers of virtual method calls, because their headers link to virtual dispatch tables. Contrarily, unboxed values cannot handle virtual dispatch:

```
val a: Int = 1       // can be unboxed
println(a.toString) // needs special treatment
```

There are two approaches to handling virtual calls: (1) the unboxed receiver can be boxed so the virtual call can be executed, or (2) if the corresponding method is final, its implementation can be extracted into a static method, rendering the call static instead of dynamic. Both of these techniques have been used in practice, although the second is markedly better for performance: in the method extraction process, the receiver becomes an explicit parameter and can be unboxed. In Scala, methods extracted from value classes are called extension methods [7]:

```
def extension_toString(i: int): String = ...
```

For the earlier example where `val c = a + b`, boxing `a` and applying the object-oriented `+` operation would be suboptimal, as it would require boxing `b` too and unboxing the result of the operation:

```
val c: int = unbox(box(a) + box(b))
```

Instead, we can use the extension method approach, rewriting the call to use the platform-intrinsic addition operation, which we denote as _+_ in the example. The intrinsic _+_ operation requires unboxed representations, so `a` can act as the receiver and `b` as the argument. Finally, the result is also unboxed, so no coercion is necessary:

```
val c: int = a _+_ b
```

3.3 Selectivity

We argue that selectivity should be built into data representation transformations as a first-class concern, allowing the programmer or the compiler to individually pick the values that will use alternative representations. Most state-of-the-art data representation transformations make the assumption that all values that can use an alternative representation should use it. However, we identified several cases that invalidate this assumption:

The low level target language may impose certain restrictions on the representations used. For example, the Scala compiler targets Java Virtual Machine (JVM) bytecode [43], which, at the time of writing, does not have a notion of structs and only allows methods to return a single primitive type or a single object. This restriction forces all methods returning multi-parameter value classes to keep the return type boxed, which is only possible if the compiler can selectively pick the values to be unboxed;

Bridge methods [19] are introduced to maintain coherent inheritance and overriding relations between generic classes in the presence of erasure and other representation transformations. Bridge methods are introduced when the low-level signature of a method does not conform to one of the base method it overrides. Consider the following example:

```
@value class D(val x: Int)
class E[T] {
  def id(t: T) = println("boo")
}
class F extends E[D] {
  override def id(d: D) = println("ok")
}
```

A naive translation, which doesn't account for erasure, will output the method `F.id` with a low-level signature `(d: int): Unit`, which, on the JVM platform, does not override the base method `E.id` with the low-level signature `(t: Object): Unit`. This will lead to virtual calls to `E.id` not being dispatched to `F.id`. A correct translation for `F` must introduce a bridge method that takes an instance of the value class `D` as an boxed argument. This method is correctly perceived as overriding `E.id` by the JVM:

```
class F extends E[D] {
  override def id(d: Object) = id(unbox(d))
  def id(d: int) = println("ok")
}
```

Generating this code is impossible if the data representation transformation always unboxes `D`, making bridge methods another example that requires selectivity.

The optimal data representation is not always unboxed. If a value is produced and consumed in its boxed representation, there is no reason to unbox it:

```
def reverse_list(list: List[Int]): List[Int] = {
  var lst: List[Int] = list
  var tsl: List[Int] = Nil
  var elt: Int = 0 // stored in unboxed form
  while (!lst.isEmpty) {
    elt = lst.head // converting boxed to unboxed
    tsl = elt::tsl // converting unboxed to boxed
    lst = lst.tail
  }
  tsl
}
```

If the data representation transformation hardcodes the fact that all primitive types should be unboxed, this code becomes very slow: during each iteration, assigning the `head` of the (generic) list to `elt` coerces a boxed integer to the unboxed representation, while the subsequent statement performs the inverse transformation, creating a new boxed integer from `elt`. This sequence of coercions not only impacts performance but also creates redundant heap garbage.

Summarizing §2 and §3, we note that an ideal data representation transformation should be smart about introducing coercions, should account for object orientation and should allow for selective coercions. The next section presents exactly that - a general, consistent, optimal, selective and object-oriented data representation transformation.

4. Late Data Layout

This section presents an approach to unifying data representation transformations under a general, consistent, optimal and selective mechanism: the Late Data Layout. We start with an overview (§4.1) and then present the three phases of the mechanism (§4.2-4.4), followed by their properties (§5).

4.1 Overview

The type-driven data representation transformation (§2.4) has shown that coercions can be guided by the type system. Still, this approach was limited by the fact that high-level concepts have to injectively map into low-level representations, which is not always the case. Furthermore, as we will see in this section, local type inference [48] is the key to optimally introducing coercions in a type-driven transformation.

Instead of directly jumping to the target representation (i.e. `int` in the examples), Late Data Layout (LDL) makes the transition in three phases: first it uses annotated types to mark the values that will use an alternative representation (the INJECT phase), then it adds coercions in places where annotation mismatches occur, signaling the incompatible representations (the COERCE phase) and finally, in the last step, it transforms annotated types to the target representation (the COMMIT phase). Using annotated types allows high-level concepts to map injectively to alternative representations, enabling type-driven transformations.

The three LDL phases are added to the compiler pipeline. The transformation expects a correct, type-checked program AST as input and outputs another correct, type-checked AST, where the high-level concept has been replaced by its representations. During the transformation, the program is type-checked again, so the type-checking procedure needs to be idempotent: once a program was successfully type-checked once, further type-checking runs should succeed and produce the same result.

A desirable property is that, given a type system with local type inference [48, 52], the LDL mechanism can optimally insert coercions, making peephole optimizations redundant. Still, to use the LDL mechanism, we need to impose two restrictions on the representation coercions:

- isomorphism of the representations: `box(unbox(t)) = t` and `unbox(box(t)) = t`;
- purity of the coercions: coercions between representations should not produce any side-effects.

Given these two restrictions, the coercions can "float" in the AST and can be optimally inserted.

Throughout the section we use the following example:

```
def fact(n: Int): Int =
 if (n <= 1)
   1
 else
   n * fact(n - 1)
```

While parsing the source code, the Scala compiler desugars this program to:

```
def fact(n: Int): Int =
 if (n.<=(1))
   1
 else
   n.*(fact(n.-(1)))
```

In the desugared version, operators are transformed into method calls, and we make this explicit by adding the commonly accepted method call notation: `receiver. method (args)`. Thus, an expression such as `n <= 1` is actually expressed as a call to the <= method: `n.<=(1)`.

The LDL mechanism consists of three phases: INJECT, COERCE and COMMIT. The next sections present each individual phase.

4.2 The INJECT Phase

The INJECT phase selectively marks values, such as fields or method arguments, with the target representation. This is done by annotating their type, for example, by adding the `@unboxed` annotation to a primitive type. The annotations can be introduced either automatically, by the compiler, based on the interactions of different language features, or manually, by programmers. This provides the flexibility necessary to capture a wide variety of transformations: some of the transformations work automatically, like unboxing primitive types and value classes, whereas others, like staging, require manual annotation, corresponding to domain-specific knowledge. In the latter case, the INJECT phase can be omitted from the compilation pipeline.

The INJECT phase transforms the running example to:

```
def fact(n: @unboxed Int): @unboxed Int =
 if (n.<=(1: @unboxed Int))
   (1: @unboxed Int)
 else
   n.*(fact(n.-(1: @unboxed Int)))
```

The constant literals were explicitly marked for unboxing: the literal constant `1` can be produced either as a boxed or unboxed value, but the unboxed representation is preferred. Therefore, constant literals are ascribed to `@unboxed Int` and, if necessary, the next phase can add a boxing coercion.

Although the example given uses a single alternative representation, this is not a requirement. For example, in the latest version of the miniboxing plugin, we use three representations: generic, miniboxed to a long integer and miniboxed to a double-precision floating point number. To encode this, we use the generic annotation `@storage[T]`. By annotating with `@storage[Long]` and `@storage[Double]` we can choose how the value will be represented. In this case, we have three coercions: `minibox2box`, `box2minibox` and `minibox2minibox`. The last coercion, `minibox2minibox`, changes the underlying miniboxed representation, either from long to double or back.

The annotations are used to carry representation information, but their underlying semantic is controlled externally, by an annotation checker, which is orthogonal to the language's type system. In a simplified view, whenever two types `T` and `S` are involved in a subtyping check, `S <: T`, two conditions are being checked: (1) that `S' <: T'` according the the standard type system, where `S'` and `T'` are `S` and `T` without any annotations and (2) that all the annotation checkers present agree that, given the annotations in `S` and `T`, they can be subtypes: `S <: T`. In reality, these two steps are made in tandem, to account for variance in generics, which relies on the sub-typing relation of the type arguments.

The transformation mechanism injects an annotation checker that allows the different representations to be compatible during the INJECT phase. This is done on purpose in the LDL mechanism, to allow the delayed introduction of coercions. Should annotated types be incompatible in the INJECT phase, the AST would become type-inconsistent, requiring the introduction of coercions to regain consistency. But there is a big win in being able to manipulate the tree with annotations but without coercions: for miniboxing, methods can be redirected to "specialized" variants without worrying about coercions, while for value classes and primitive unboxing, bridge methods can forward to their target without explicitly coercing the arguments.

In the next phase, the annotation checker makes representations incompatible, driving the introduction of coercions.

4.3 The COERCE Phase

The COERCE phase is the centerpiece of the LDL mechanism and is similar for all data representation transformations. It is responsible for introducing the necessary coercions such that representations are used consistently in the transformed program. Unlike the INJECT phase, which updates the signatures of symbols, the COERCE phase only adapts the AST by introducing coercions, based on the additional representation information carried by annotated types.

The COERCE phase transforms the abstract syntax tree in two steps: (1) in the annotation checker, the different representations become incompatible, thus invalidating the current AST and (2) the COERCE phase type-checks the AST and introduces coercions where necessary. The coercions are introduced based on the representation mismatches revealed by the local type inference (§4.3.1): when a certain representation is required but a different one is passed, a coercion is introduced (§4.3.2). Since the names have been resolved and the tree has been type-checked before, type-checking the tree again will only be responsible for inserting coercions (given that the type checker is indeed idempotent). Finally, object-oriented features of the language need to be taken into account (§4.3.3).

4.3.1 Local Type Inference

Local type inference [48, 52] is used to reduce the boilerplate in source code, by inferring certain type annotations instead of requiring the programmer to write them by hand.

Type inference is done in two steps: (1) creating synthetic type variables for polymorphic expressions in the AST and (2) using bidirectional propagation to gather constraints on the synthetic type variables, which are then solved to exact types. We will illustrate how it works with an example:

```
def identity[T](t: T): T = t
identity(3) // should infer identity[Int](3)
```

Since the call to `identity` is polymorphic, the local type inference algorithm introduces a synthetic type variable, which we call `?T` in the example:

```
identity[?T ](3)
```

It then type-checks the AST using bidirectional propagation. Along with propagating types from the innermost AST nodes to the outside, local type inference also propagates expected types from the outside nodes towards the inside. Namely, in the example, `identity[?T]` expects an argument of type `?T`, so the literal constant `3` is type-checked with an expected type `?T`. But the literal constant is known to be of type `Int`. In this case, the condition for successfully calling the `identity` method is that `Int <: ?T`. Therefore the only constraint on `?T` is that it needs to be a super type of `Int`. Solving this constraint to the most specific type yields `?T = Int`, which is replaced in the original call:

```
identity[Int](3)
```

In the COERCE phase we only use the expected type propagation feature, as the input AST is already type-checked and all type annotations have already been inferred. The next part describes exactly how the expected type propagation drives the introduction of representation coercions.

4.3.2 Placing Coercions

Coercions are introduced when an AST node's representation doesn't match the one required by the outside node. In the compiler, name resolution is effectively the high-level equivalent of a forward data flow analysis [20], tracking the reaching definitions via symbols. Coupled with the type system, name resolution propagates the types of symbols in a program's syntax tree and, along with them, the representation information. On the other hand, the expected type propagation in local type inference acts as a backward data flow analysis tracking the expected representation of a node.

Therefore, name resolution and local type inference collaborate to produce a forward and backward data flow analysis which detects mismatching representations:

```
def foo(x: Int): @unboxed Int =
  x // forward analysis: name "x" refers to
    //   argument x of method foo of type Int
    // backward analysis: the return type of
    //   method foo needs to be @unboxed Int
    // representation mismatch => coercion
```

AST nodes such as conditional expressions and blocks have very interesting behaviors when it comes to expected type propagation: an `if` conditional propagates the expected type to its `then` and `else` expressions while a block propagates the expected type only to its expression (the last expression in the block, the first $n-1$ expressions are treated as statements). On the other hand, since the statements in a block are designed to perform side-effecting actions and their results are ignored, they are type-checked without an expected type, thus accepting any representation.

Propagating expected types delays the introducing the coercions until a node with a fixed type is encountered, such as the value x in the previous example, and the expected type requires a different representation. This sinks coercions as deep in the AST as possible, side-stepping the need for a peephole optimization (§2.3) and making the coercion insertion optimal. Optimality is further discussed in §5.3.

Implicit conversions in the Scala programming language could also be used to introduce coercions. Both implicit conversions and representation coercions adapt a node to the type expected by the outer expression. However, since implicit conversions can be influenced by the program code, we prefer to use a separate, albeit similar mechanism to introduce the coercions, in order to avoid any interactions. The fact that implicit conversions are resolved in the compiler frontend does help: by the time LDL-based transformations kick in, implicits have been resolved, so the transformation only needs to add representation coercions.

4.3.3 Object-Oriented Aspects

During the transformation, which type-checks the AST again in a DFS approach, the COERCE phase needs to take care of the object-oriented aspects in the language. For example, method calls with unboxed receivers require either boxing or forwarding to an extension method [7]. Fortunately, super types do not require special handling: only types that can be unboxed are annotated, not their super types, so expressions that conform to super types are automatically boxed through annotation-driven coercions.

Forwarding to an extension method or intrinsic deserves a more detailed explanation. In the factorial example we use the * operator, which requires boxing the receiver and the argument and returns a boxed result. Instead of the * operation, the COERCE phase will use _*_, the platform-provided intrinsic multiplication for unboxed integers. To do so, while descending in the AST to type-check each node, the COERCE phase intercepts method calls where the receiver is unboxed. One such method is n.*(...), where n has type @unboxed Int. Since the * operation does have an intrinsic equivalent, _*_, it is replaced in the tree. Following the replacement, the COERCE transformer descends and type-checks the argument with the new expected type, which requires it to be unboxed. Once the argument has been type-checked, the COERCE transformer returns to the intrinsic method call, and, given the expected type for the result, decides whether a coercion is necessary or not. The result is:

```
def fact(n: @unboxed Int): @unboxed Int =
  if (n._<=_(1: @unboxed))
    (1: @unboxed)
  else
    n._*_(fact(n._-_(1: @unboxed)))
```

No coercions are introduced at all, but the operators are now redirected to their intrinsic variants _<=_, _*_ and _-_.

4.4 The COMMIT Phase

The COMMIT phase is the final phase in the transformation mechanism and is meant to transform the annotated types to the actual alternative representation. It is also tasked with replacing coercion markers (box and unbox) by the actual operations necessary for creating objects and extracting the unboxed values. For instance, when unboxing primitive types, the COMMIT phase is going to transform @unboxed Int to int, unbox into a method call that returns the unboxed value, and box into the construction of a java.lang.Integer object. If extension methods were used (in this case the underlying platform's intrinsics), their signatures are automatically transformed to the native representation (i.e. replacing @unboxed Int by int). After the COMMIT phase the program is fully transformed:

```
def fact(n: int): int =
  if (n._<=_(1))
    1
  else
    n._*_(fact(n._-_(1)))
```

The COMMIT phase is heavily dependent on the transformation at hand when updating the symbol signatures and the AST. For certain transformations, it can go beyond replacing coercion markers by actual operations: unboxing multiple-parameter value classes requires creating multiple fields and populating them. Yet, the AST transformations have local scopes and are always triggered either by a coercion marker, an annotated type in the node or a library method that carries special semantics for the given transformation. For example, in the staging plugin, the method compile[T](expr: @staged T): T has the special meaning that a staged expression needs to be compiled to optimized code and executed. It is redirected by the staging plugin from identity (the default implementation, in the case staging is turned off) to a special implementation that generates the code, compiles it and invokes the result. The Validation section of the paper (§6) describes the rules of the COMMIT phases for each of the three extensions we developed using the Late Data Layout mechanism.

5. Transformation Properties

This section presents the properties of the Late Data Layout mechanism. Although a partial formal description of the transformation is available [73], this section only provides an intuitive reasoning about the properties of the mechanism:

- consistency in terms of value representation;
- selectivity in terms of value representation;
- optimality in terms of runtime coercions;

To the best of our knowledge, we are the first to describe a general-purpose mechanism that has the last two properties: selectivity and optimality.

5.1 Consistency

In the LDL mechanism, we track the representation of each value, inside its type. During the COERCE phase, the annotation checker makes the representations incompatible, leading to the introduction of coercions, so the tree type-checks successfully. Since type-checking builds a formal proof of the program correctness modulo the theory introduced by types, injecting the representation information into the type system allows it to extend the correctness proof to the consistency of representations and coercions. This leads to the property that trees transformed by the coerce phase are consistent in terms of representation.

It worth observing that, depending on the transformations in the COMMIT phase, a consistent program may become inconsistent. This only occurs because the mechanism is general-purpose, so it does not impose the actions performed in the COMMIT transformation. Still, for simple transformations, where annotated types are transformed to another representation along with their coercions, the consistency guarantee extends to the entire transformation. On the other hand, for complex transformations, such as the ones necessary for multi-parameter value classes, each individual rewriting rule has to be proven correct. Still, it is important that coercions are introduced consistently and optimally, allowing the COMMIT transformation to build on a solid foundation and to have a simplified proof based on the LDL invariants.

5.2 Selectivity

Selectivity results directly from the fact that individual values can have their types annotated separately. Furthermore, the miniboxing plugin demonstrated that the LDL mechanism can handle multiple representations without any issues.

5.3 Optimality

Experience with the LDL mechanism reveals an interesting fact: Thanks to the expected type propagation in local type inference, representation constraints are propagated deeper in the AST and, in certain branches or expressions, the coercions are elided completely, when the expected representation matches the actual one. This leads us to think that, for any given execution trace of the input program, the LDL mechanism minimizes the number of coercions executed. While we do not formally prove this property, we give an intuitive explanation of why it occurs. It should be noted that the minimization is done modulo the annotations introduced by the INJECT phase, that dictate which values are unboxed and that can potentially be suboptimal (§3.3).

Revisiting the behavior of `if` nodes and blocks, described in Section 4.3.2, we can partition the AST nodes into `opaque` and `transparent`. Opaque AST nodes have a fixed type, which is not influenced by the expected type of the outer expressions. For example, a constant literal `3` is an integer regardless of the expected type. Transparent nodes, on the other hand, adapt to the expected type by further constraining their children AST nodes, as the `if` expression does. This binary classification does not capture the full wealth of features in Scala's type checker, such as implicit conversions, overloads and polymorphic nodes. However, these are typically resolved during the initial program type-checking phase, in the compiler frontend, and do not influence LDL-based transformations.

Furthermore, the relation between an AST node and its child sub-nodes can be characterized as either `oblivious` or `constraining`. The typical example of oblivious relation occurs between blocks and the statements they contain: the results produced by the statements are ignored, so there is no reason to constrain them. Contrarily, the constraining relationship propagates an expected type to the subnodes. Refining this further, we have propagated and fixed constraints. For example, the condition of an `if` expression has a fixed constraint that it needs to be a boolean. On the other hand, the `then` and `else` branches, have propagated constraints: they get the expected type from the parent node.

With these definitions, we can observe that the peephole optimization actually implements the transport of coercions through transparent nodes with propagated constraints and the removal of constraints from oblivious nodes. The similarity between an eager transformation with a peephole optimization and the LDL mechanism is now becoming clear: the peephole optimization is for coercions what the local type inference is for expected types: a mechanism for transporting information in the AST which sinks either coercions or constraints deeper into the tree.

Thanks to expected type propagation, when a coercion is introduced, it is introduced as deep in the tree as possible, even if this requires duplication. Let us take an example:

```
def baz(t1: @unboxed T, t2: @unboxed T, t3: T,
    c1: Boolean, c2: Boolean): @unboxed T =
  if (c1)
    t1
  else
    if (c2)
      t2
    else
      unbox(t3)
```

We can see that only `t3` is coerced, since the `if` expressions are transparent. During execution, sinking coercions in the tree means they are only executed if this is unavoidable, as a representation mismatch occurred at one point in the execution trace. An interesting remark is that a minimum number of constraints in any execution trace doesn't translate to a minimum total number of constraints introduced in the program:

```
def buz(t1: T, t2: T, c: Boolean): @unboxed T =
  if (c)
    unbox(t1)
  else
    unbox(t2)
```

Since constraints are sunk to the bottom of the tree, they may be duplicated several times for nodes such as conditionals and pattern matches. Therefore, the total number of coercions introduced in the tree is not minimum, in our example being 2, instead of 1, which corresponds to coercing the `if` expression. Still, given any execution trace in the program, the total coercions executed is minimum, in our example, just 1. Note that coercions may be further reduced or increased by changing the output of the INJECT phase (§3.3). Also, naively implementing the COMMIT phase can introduce to redundant coercions. Unfortunately, it is impossible to reason about the INJECT and COMMIT phases in a general setting, as they are specific to each transformation.

Arguably, sinking coercions could potentially place them inside hot loops. In Scala, since `for` loops are desugared to method calls, the only two mechanisms for low-level looping are `while` loops and tail-recursive calls. Both `while` loops and method calls are opaque nodes in the AST and do not propagate expected types. Therefore, for the Scala ASTs, we do not expect the LDL mechanism to sink coercions inside hot loops. Still, coercions may be introduced in hot loops based on the annotations introduced by the INJECT phase for loop-local values, which may require coercing (§3.3).

6. Validation and Evaluation

This section describes how we validated the Late Data Layout mechanism by using it to implement three very different language features: value classes, specialization via miniboxing and support for multi-stage programming.

In our case studies we observed increased productivity thanks to the reuse of the Late Data Layout mechanism. Two decisions in LDL also provided tangible benefits to the development process: (1) decoupling the decision to unbox values from the mechanism that introduces coercions and (2) decoupling the alternative representation semantics from the coercions and annotated types.

A highlight of the validation is the fact that we reimplemented and extended the Scala compiler support for value classes [7] with just two developer-weeks of work and without reusing any pre-existing code.

We begin by describing the plugin architecture in the Scala compiler and how it can be used to implement data representation transformations. Afterwards, we present and evaluate each of the three case studies.

6.1 Scala Compiler Plug-ins

The Scala compiler allows extension via plugins. These can customize the type system through annotation checkers and can inject new compilation phases. In this section we describe the annotation checker framework and the custom compiler phases added by the LDL mechanism.

The annotation checker framework allows compiler plugins to inject annotations during type-checking, to provide custom logic for the joins and meets of annotated types and to apply custom transformations to abstract syntax trees (ASTs) whose type is annotated. Still, the most important feature for the LDL transformation is allowing plugins to extend the vanilla subtyping logic in the Scala compiler by providing custom and phase-dependent rules for annotated types. Using this framework, *Rytz* created a purity and effects checker [60] that uses annotations to track side-effecting code, while *Rompf* implemented a type-driven continuation-passing style (CPS) transformation [56].

LDL-based transformations use the annotation checker framework to encode the high-level concept with its representations in the type system. Before the COERCE phase, annotated types are compatible with their non-annotated counterparts, exposing the unified concept. During and after the COERCE phase, however, this compatibility is broken, emulating the difference between representations. This newly created incompatibility drives the insertion of coercions in the program's abstract syntax tree (AST).

```
def annotationsConform(tpe1: Type, tpe2: Type) =
 if (phase.id < coercePhase.id)
  true
 else
  // this check can be expanded to account
  // for multiple representations, not just
  // unboxed or boxed, which corresponds to
  // annotated or not annotated:
  (tpe1.isAnnotated == tpe2.isAnnotated) ||
    tpe2.isWildcard
```

Custom compiler phases allow plugins to transform both the AST and the symbol signatures at precise points in the compilation pipeline. An LDL-based plugin typically adds three custom phases, corresponding to INJECT, COERCE and COMMIT. However, each specific transformation is free add more phases and can even interpose them between the standard LDL phases.

The INJECT phase initiates the transformation process by marking values with their alternative representations. To do so, the phase visits all entries in the symbol table and updates their signatures: fields, local values, method arguments and returns are marked using annotated types. Since this phase is dependent on the transformation and typically does more than just adding annotations, it will be described in detail in each of the case studies.

The COERCE phase is the core of the transformation mechanism and is similar for all case studies. Since the annotation checker exposes the different representations, the COERCE phase essentially starts with an inconsistent abstract syntax tree, where the type mismatches correspond to clashing representations. The COERCE phase makes the tree consistent again by type-checking it while using local type inference to guide the introduction of coercions.

In Scala, the type checker consists of two parts:

1. a typing judgement, which assigns a type to each AST node and
2. an adaptation routine, which transforms AST nodes so their type matches the expected one.

The adaptation routine is responsible for inserting implicit conversions, resolving implicit parameters and synthesizing reified types [62]. The next code snippet shows a heavily simplified Scala-like type-checking algorithm:

```
def typed(tree: Tree, exp: Type): (Tree, Type) =
  /* (1) */ typing_judgement(tree, exp) match {
    case (tree1, tpe1) if
         subtype(tpe1, exp) &&
         annotationsConform(tpe1, exp) =>
      (tree1, tpe1)
    case (tree2, tpe2) =>
      /* (2) */ adapt(tree2, tpe2, exp)
  }
```

We assume the methods have the following signatures:

```
def typing_judgement(tree: Tree, exp: Type):
                                (Tree, Type) = ...
def adapt(tree: Tree, tpe: Type, exp: Type):
                                (Tree, Type) = ...
```

In the type-checking algorithm, the adaptation routine (2) is only triggered if the type of the current tree, as decided by the typing judgement (1), does not conform to the expected type. As a result, only opaque nodes (§5.3) reach the adaptation routine. For example, the typing judgement for an `if` expression will propagate the expected type to the branches, leading to each individual branch conforming or being adapted to conform. This makes the conditional itself conform, therefore bypassing adaptation.

The main change added by the COERCE phase to the typing algorithm concerns the adaptation routine: whenever a mismatch between representations is detected, a coercion is introduced. For example, if the expected type is `Int`, and the actual type is `@unboxed Int`, a `box` coercion is added.

The COERCE phase also adds a rule to the typing judgement: when a method call is encountered, the receiver expression is type-checked without an expected type, in order not to constrain it. If the result is a boxed expression, the method call can be performed as-is. On the other hand, if the result uses an alternative representation, there are two options: (1) if the specific transformation does have alternative methods for unboxed receivers (such as extension methods), the call can be redirected to the alternative method or (2) if such a method is not available, the receiver expression is type-checked again expecting a boxed type, leading to the introduction of a coercion. This allows performing method calls regardless of the receiver's representation.

Thanks to the annotation checker, when a node is type-checked expecting a super type, it is automatically boxed. This occurs because, as discussed in §3.1, super types of an unboxed type cannot themselves be unboxed. Along with the method call transformation, the super type boxing forms the LDL support for the object-orientated features in the Scala programming language.

Finally **the COMMIT phase** transforms the symbol signatures and the tree to use the low-level alternative representations. When the AST reaches the COMMIT phase, it is consistent and has the all the annotations and coercions necessary to guide the transformation. Again, since this phase is specific to the representation, we describe it the each of the case studies, along with counting the lines of code and the number of rewrite rules.

6.2 Case Study 1: Value Classes

Value classes [2, 7, 29] marry the homogeneity and dynamic dispatch of classes with the memory efficiency and speed of C-like structures. In order to get the best of both worlds, value classes have two different in-memory representations. Instances of value classes (referred to as value objects) can be represented as fully-fledged heap objects (the boxed representation) or, when possible, use a struct-like unboxed representation with by-value semantics.

For instance, in the example below, the `Meter` value class is used to model distances in a flexible and performant manner, providing both object-orientation (including virtual methods and subtyping) and efficiency of representation. Our implementation transforms methods +, <= and `report` such that their arguments and return types are unboxed value classes. Furthermore, values of type `Meter` will use the unboxed representation wherever possible.

```
@value class Meter(val x: Double) {
  def +(other: Meter) = new Meter(x + other.x)
  def <=(other: Meter) = x <= other.x
}
def report(m: Meter) = {
  if (m.<=(new Meter(9000)))
    println(m.toString)
}
```

Before we dive into the transformation, let us consider some basic facts about value classes, correlating them with existing implementations for C# [2] and Scala (both the official transformation shipped with Scala 2.10 [7], and the prototype we present in this paper).

Final semantics. Even though value classes can extend traits, their participation in the class hierarchy has to be limited in order to allow correct boxing and unboxing. Indeed, if along with `Meter` it were possible to define another value class `Kilometer` that extended `Meter`, then unboxing `m` would be ambiguous, as its boxed representation might be either of the classes. This observation is consistent with both C#, where value classes cannot be extended, and Scala, where value classes are declared by inheriting from the marker class `AnyVal` and are automatically made final.

By-value semantics. When compiling value classes to low-level bytecode, additional care must be taken to accommodate their by-value semantics on otherwise object-oriented platforms: both the JVM and the CLR have a universal superclass called `Object` that exposes by-reference equality and hashing. Moreover, both platforms provide APIs to lock on objects based on reference. While we can't control what happens to value objects that are explicitly cast to `Object`, we can restrict uses of by-reference APIs. In C# this is done by having a superclass of all value classes, called `ValueType`, which provides reasonable default implementations of `Equals` and `GetHashCode`, whereas in Scala all value classes get `equals` and `hashCode` implementations generated automatically. Both in C# and Scala synchronization on value classes is outlawed.

Single-field vs multi-field. While single-field value classes like `Meter` trivially unbox to a single value, devising an unboxed representation for multi-field value classes may pose a challenge if the underlying platform does not provide support for structures. And indeed, in the case of Scala, the JVM does not support structs or returning multiple values, so we have to box multi-field value objects when returning them from methods. Still, for fields, locals and parameters we do unbox multi-field value objects into multiple separate entries, providing a faithful emulation of struct behavior. It is worth noting that the value class implementation in Scala 2.10 only supports single-field value classes, therefore sidestepping this issue altogether. C# doesn't have this problem, because the .NET CLR provides a primitive for structs.

Having seen these aspects of value classes, we can now dive into the implementation of our prototype. It follows the standard three phases: INJECT, COERCE and COMMIT, all preceded by an extension methods phase, ADDEXT:

The ADDEXT phase makes several changes to the tree: it adds standard `hashCode` and `equals` implementations for value classes, it transforms value class methods into extensions and finally adds redirects from the value class to the extension methods in the companion object. The extension methods are later used by the COERCE phase, which redirects method calls as described in §4.3. The result is:

```
@value class Meter(val x: Double) {
 def +(other: Meter) = Meter.+(this, other)
 def <=(other: Meter) = Meter.<=(this, other)
 ... // hashCode, equals redirections
}
object Meter {
 def +(self: Meter, other: Meter) =
  new Meter(self.x + other.x)
 def <=(self: Meter, other: Meter) =
  self.x <= other.x
 ... // hashCode, equals extension methods
}
def report(m: Meter) = ...
```

The INJECT phase marks values to be transformed using the `@unboxed` annotation. It marks all fields, locals and parameters of value class type as well as return types of methods that produce single-field value objects:

```
@value class Meter(val x: Double) {
 def +(other: @unboxed Meter) =
  Meter.+(this, other)
 def <=(other: @unboxed Meter) =
  Meter.<=(this, other)
 ... // hashCode, equals redirections
}
object Meter {
 def +(self: @unboxed Meter, other: @unboxed
    Meter) = new Meter(self.x + other.x)
 def <=(self: @unboxed Meter, other: @unboxed
    Meter) = self.x <= other.x
 ... // hashCode, equals extension methods
}
def report(m: @unboxed Meter) = {
 if (m.<=(new Meter(9000)))
  println(m.toString)
}
```

This is a notable use-case for the first-class selectivity support provided by the LDL mechanism. Methods that return multi-field value objects are not annotated with `@unboxed` on the return type, since the JVM lacks the necessary support for multi-value returns: Simply leaving off the `@unboxed` annotation is all it takes to have the result automatically boxed in the method and unboxed at the caller.

Another responsibility of the INJECT phase is the creation of bridge methods (§3.3). If a method that has value class parameters overrides a generic method, INJECT creates a corresponding bridge:

```
trait Reporter[T] {
 def report(x: T): Unit
}
class Example extends Reporter[Meter] {
 def report(x: Meter) = report(x) // bridge
 override def report(x: @unboxed Meter) = ...
}
```

Code emitted for these bridges is particularly elegant, again thanks to the selectivity of the transformation. It turns out that it is enough to just have the bridge be a trivial forwarder to the original method with its parameters being selectively annotated. This produces a compatible signature for the JVM and the COERCE phase automatically manages representations by introducing coercions. Dziakuj LDL!

The COERCE phase follows the pattern established in §4, making `@unboxed` types incompatible with their non-annotated counterparts and inserting `box` and `unbox` markers in case of representation mismatches. The coerce phase also redirects to extension methods where possible. For our running example, the following code is produced:

```
@value class Meter(val x: Double) {
 def +(other: @unboxed Meter) =
  Meter.+(unbox(this), other)
  Meter.<=(unbox(this), other)
 ... // hashCode, equals redirections
}
object Meter {
 def +(self: @unboxed Meter, other: @unboxed
    Meter): @unboxed Meter = unbox(new
    Meter(box(self).x + box(other).x))
 def <=(self: @unboxed Meter, other: @unboxed
    Meter) = box(self).x <= box(other).x
 ... // hashCode, equals extension methods
}
def report(m: @unboxed Meter) = {
 if (Meter.<=(m, unbox(new Meter(9000))))
  println(box(m).toString)
}
```

The COMMIT phase uses the annotations established by the INJECT phase and the marker coercions to represent the annotated value classes by their fields. In particular, the COMMIT phase changes the signatures of all fields, locals and parameters annotated with `@unboxed` into their unboxed representations, creating as many duplicated fields as necessary to store the unboxed multi-field value classes. Return types of methods are unboxed as well, but only for single-field value classes.

On the level of terms, the transformation centers around the coercion markers, causing `box(e)` calls to become object instantiations and rewriting `unbox(e)` calls to field accesses. Additionally, we devirtualize `box(e).f` expressions as much as possible, which is done by transforming `box(e).f` into a reference to the unboxed field.

Finally, term transformations perform the necessary bookkeeping to account for duplicated fields (arguments and parameters of value class types are duplicated as necessary, assignments to locals and fields or value class types become multiple assignments to duplicated locals and fields, etc).

The COMMIT phase transforms our example to:

```
final class Meter(val x: Double) {
  def +(other: Double) = Meter.+(x, other)
  def <=(other: Double) = Meter.<=(x, other)
  ... // hashCode, equals redirections
}
object Meter {
  def +(self: Double, other: Double): Double =
      self + other
  def <=(self: Double, other: Double) = self <=
      other
  ... // hashCode, equals extension methods
}
def report(m: Double) = {
  if (Meter.<=(m, 9000))
    println(new Meter(m).toString)
}
```

It is worth mentioning that even with the necessity to cater for the lack of built-in struct support in the JVM, the resulting transformation is remarkably simple. First, we have been able to implement it without changing the compiler itself (in particular, without customizing the built-in ERASURE phase). Second, custom logic in INJECT, COERCE and COMMIT phases spans only about 500 lines of code. This shows the LDL mechanism can significantly reduce the effort necessary to implement complex data representation transformations.

6.2.1 Evaluation

We evaluate the plugins on three metrics:

- Lines of code and complexity of the commit phase;
- Runtime performance improvements;
- Additional features added to the LDL mechanism.

Lines of code and complexity. The value class plugin has 17 files Scala files with 1286 lines of code, as reported by the cloc counter [1]. Unfortunately, it is impossible to compare these stats to the Scala implementation, as several transformations are merged into the ERASURE phase and untangling them is a very difficult challenge.

The COMMIT phase for the value class plugin has 180 lines of code and 29 transformation rules:

- 6 rules for transforming coercions;
- 23 rules for different AST nodes - triggered either by coercions or by annotations.

In the COMMIT phase, many of the rules that expand definitions into multiple fields are triggered either by coercions or by annotations, such as `@unboxed` on the value definition:

```
val c: @unboxed Complex = ...
//      => will be split into c_re and c_im.
```

Runtime performance. We evaluated the runtime performance using an FFT example from the Rosetta Code website [6]. The speedups we observed come from transforming the complex number case class into a value class, allowing it to be inlined. The results we obtained using the scalameter [53] benchmarking framework, expressed in milliseconds, were:

```
::Benchmark FFT.Scala Complex::
Parameters(data size = 2^ -> 4): 11.9418295

::Benchmark FFT.Valium Complex::
Parameters(data size = 2^ -> 4): 11.8187571
```

The speedup is only 1% because, at this point, we cannot unbox value classes when returning them. We are currently looking at different ways to improve the performance by side-effectfully writing the value to a thread-local variable on method return and reading it back in the caller.

A different benchmark we tried was adding up 2^{14} complex numbers:

```
::Benchmark Ops.Scala Complex::
Parameters(data size = 2^ -> 14): 0.1461588

::Benchmark Ops.Valium Complex::
Parameters(data size = 2^ -> 14): 0.0930053
```

This is where value classes really speed up the program: a simple `@value` annotation produces an almost 2x speedup.

The extra feature added by the value class plugin over the standard LDL mechanism is the ability to indicate code patterns that should always be boxed. This is done in the COERCE phase and it reduces the code patterns the COMMIT phase needs to handle. This feature requires an extra 3-line rule in the typing judgement which matches a pattern and type-checks the expression with a boxed expected type. In the current implementation, the pattern matches unstable expressions (that can change the value from one access to the next), which cannot be unboxed to multiple fields:

```
val c1: @unboxed Complex = ...
val c3: @unboxed Complex = ...
val c3: @unboxed Complex =
 unbox(if (...) // if => unstable expression
          box(c1)
        else
          box(c2))
```

The COERCE phase requires the `if` expression to be boxed and unboxes it before assigning the result to `c3` (since `c3` is unboxed, we assign to the duplicated fields of `c3`: `c3_re` and `c3_im`). There are three reasons for this transformation: (1) to reduce the commit phase complexity, (2) since the Scala AST representation does not allow multi-field block returns and (3) since this pattern is easily detected and optimized by the just-in-time compiler in the JVM.

6.3 Case Study 2: Miniboxing

The miniboxing transformation [5, 75] is the most complex case study and also the most established, being under development for almost two years. The miniboxing plugin initially used an eager transformation coupled with a peephole optimization. The difficulties in maintaining and expanding the peephole rewriting rules motivated the development of the LDL mechanism. This section briefly mentions the ideas behind specialization and miniboxing and then explains how the code is transformed using the three phases of the LDL mechanism: INJECT, COERCE and COMMIT.

Specialization [21] improves the performance of erased generics by duplicating methods and classes and adapting them for each primitive type. These adapted versions, also called specialized variants, receive and return unboxed primitive types, thus allowing the program to use them efficiently. Yet, specialization leads to bytecode duplication, with 10 variants per type parameter: 9 for the primitive types in Scala plus the erased generic. This means that specializing a tuple of 3 elements, which has 3 type parameters, produces 10^3 classes, too much for practical use.

Miniboxing [75] was designed to reduce the bytecode explosion in specialization. It is based on two key insights: (1) in Scala, any primitive type can be encoded in a long integer, thus reducing the duplication to two variants per type parameter and (2) the encoding requires provenance information, namely a type tag that represents the original type of the long-encoded value. With miniboxing, fully specializing a 3-element tuple creates 8 classes and an interface.

To explain how the miniboxing transformation works, let us use the identity example again:

```
def identity[@miniboxed T](t: T): T = t
identity[Int](5)
```

The @miniboxed annotation on the type parameter T triggers the transformation of the the method. This will duplicate and adapt the body of identity, creating a new method identity_M, care acceptă primitive. This new method encodes the primitive types into a long integer and requires a type tag corresponding to the type parameter T. The low level code resulting from compilation is:

```
def identity(t: Object): Object = t
def identity_M(tag: byte, t: long): long = t
_identity_M(INT, int2minibox(5))
```

Let us walk through the steps necessary to obtain this low level code. The first step of the **INJECT phase** duplicates the method identity to identity_M and adds the type tag:

```
def identity[T](t: T): T = t
def identity_M[T](tag: Byte, t: T): T = t
```

In the second step, in order to adapt identity_M to primitive types, the miniboxing plugin transforms all values of type T to Long. Doing this transformation consistently and optimally requires an LDL cycle, so the INJECT phase starts by marking values of type T that will use the miniboxed encoding. The annotation used in miniboxing is @storage:

```
def identity[T](t: T): T = t
def identity_M[T](tag: Byte, t: @storage T):
    @storage T = t
```

In the third step, the INJECT phase specializes method calls. It does so by redirecting calls from miniboxed methods to their specialized variants, based on the type arguments:

```
identity_M[Int](INT, 5)
```

The COERCE phase contains the standard LDL logic. In our example, it does not change the two method definitions, but the call to identity_M gets the argument coerced (we assume the call is in statement position, otherwise the result would also have to be coerced back to Int):

```
identity_M[Int](INT, marker_box2minibox(5))
```

The COMMIT phase converts @storage T to Long and replaces the marker_ methods by their actual implementations, either the more general minibox2box / box2minibox, which use the type tag, or the more efficient minibox2X / X2minibox when X is a primitive type. The result after the COMMIT phase is:

```
def identity[T](t: T): T = t
def identity_M[T](tag: Byte, t: Long): Long = t
identity_M[Int](INT, int2minibox(5))
```

Finally, as this code passes through the Scala compiler's backend, the ERASURE phase unboxes the Long integers into long and erases the type parameter T to Object. This produces the exact result we showed in the beginning.

It is worth mentioning that miniboxing exploits all the flexibility available in the LDL mechanism: in the last version it features 2 alternative encodings (miniboxing to Long or Double), the alternative representation mapping is not injective, since all miniboxed type parameters map to either Long or Double, the selectivity is used to generate bridge methods for similar reasons to those presented in §3.3 and the compatibility between annotated and non-annotated types in the INJECT phase is used to easily redirect method calls from miniboxed methods to their specialized variants.

6.3.1 Evaluation

Lines of code and complexity. The miniboxing plugin has 17 Scala files with 2584 lines of code. The specialization transformation currently available in the Scala compiler [21] has 2 Scala files with 1541 lines of code. However, we are not comparing similar things: the miniboxing plugin performs a more complex transformation compared to specialization and bears the boilerplate necessary to build a compiler plugin.

The COMMIT phase for miniboxing has 260 lines of code and 12 transformation rules:

- 3 rules for coercions (minibox2box, box2minibox, minibox2minibox);

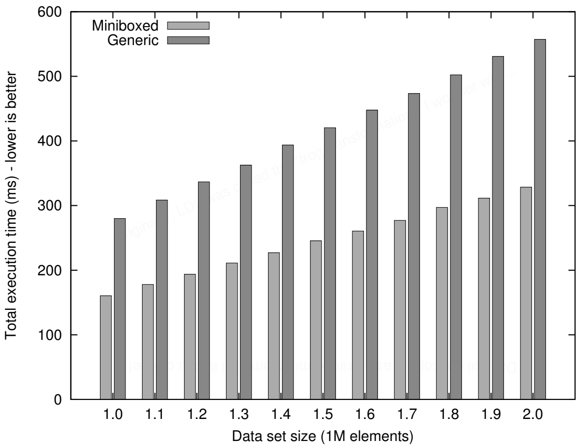

Figure 1. Least squares method using linked lists

- 4 rules for redirecting methods inherited from `Any`, such as `toString` - triggered by coercions;
- 4 rules for optimizing arrays - triggered by array operations (`a.{apply,update,length}` and `new T[]`);
- 1 extra rule for optimizing the function representation.

In the miniboxing plugin, universal methods inherited from `Object` are redirected to library-provided extension methods, and, since they do not require a different representation, the redirection is done in the COMMIT phase instead of the COERCE phase. These rewritings could have been done in the COERCE phase equally well.

We can compare the miniboxing plugin before and just after the LDL mechanism was added:

- Before (29th of October 2014): 2285 LOC (out of which approximately 500 LOC in the peephole optimization)
- After (14th of February 2014): 2246 LOC (out of which approximately 200 LOC in the commit phase + 250 LOC for the general and reusable LDL mechanism)

Runtime performance. Since the miniboxing plugin has been around for some time, its runtime performance has been thoroughly benchmarked [75]. The most recent result is a benchmark on a slice of the Scala collections library [26] centered around the linked list collection. The benchmark consists of running the least squares method for fitting data points on several input sizes. The results, summarized in Figure 1, show a 45% speedup produced by using the miniboxing transformation. It should be noted that Scala collections are notoriously hard to transform, since they use many advanced features of the language, such as type classes, higher-kinded types and anonymous and nested classes. Indeed, we also tried to run the benchmark with the current specialization transformation in the Scala compiler [21], but the results were disappointing: due to technical difficulties, the specialized linked list was slower than the generic one.

The miniboxing plugin [5] has also transformed larger projects, with spire [49] being the largest at 31KLoC, and produced reliable results. This shows the LDL mechanism is not just a toy but can correctly transform large code bases.

Two extra features are added by the miniboxing plugin over the standard LDL mechanism:

- using multiple alternative representations, `Long` and `Double` in the current version. To implement this, the `@storage` annotation was parameterized with a type, allowing the INJECT phase to include the target representation in the annotation: `@storage[Long]` → `Long` and `@storage[Double]` → `Double`. This lead to a third coercion marker, `marker_minibox2minibox`;
- a second LDL cycle is used to change the object-oriented representation of functions to a miniboxing-friendly representation.

These additions are described on the miniboxing website [5].

6.4 Case Study 3: Staging

Multi stage programming [70] allows a program to execute in several steps, at each step generating new code, compiling and then executing it. In Scala, this technique has been used by *Rompf* to develop the lightweight modular staging (LMS) framework [55, 57], which removes the cost of abstraction in many high-level embedded DSLs [10, 37, 54, 68, 74].

Using the LMS framework requires the ability to lift built-in language constructs, such as method calls, `if` expressions and variable accesses. This is done by transforming these constructs into calls to methods provided by the programmer or by the LMS framework. Currently, lifting is done using a custom version of the compiler, dubbed scala-virtualized [45] or using Yin-Yang [34], a macro-based frontend that allows selectively lifting parts of a program.

In this section, we show that lifting can be modelled as a data representation transformation, allowing LDL-transformed programs to be optimized by an LMS-like framework. One of the early examples of staging given by *Rompf* is eliminating the recursion from a power function:

```
1  def pow(b: @staged Double, e: Int): @staged
     Double =
2   if (e == 0) 1.0
3   else if (e % 2 == 1) b * pow(b, e-1)
4   else {
5     val x = pow(b, e/2)
6     x * x
7   }
8  val pow5 = function(arg => pow(arg, 5))
9  println("3.0^5 = " + pow5(3.0))
10 println("4.0^5 = " + pow5(4.0))
```

The `pow` method computes b^e. The base, `b`, and the return type are marked as `@staged`, whereas the exponent, `e`, is not. This means that calls to `pow`, instead of computing a value, accumulate the operations necessary to produce b^e for a variable base `b` and a fixed exponent `e`.

Indeed, the call to `function` in line 8 first triggers the execution of `pow` for the variable base `b=arg` and the fixed exponent `e=5`. The operation graph recorded corresponds to $\mathtt{arg}^5$ and is used by the `function` call to generate optimized code, compile it, and to expose it as a function from `Double` to `Double`, corresponding to $\mathtt{arg} \Rightarrow \mathtt{arg}^5$:

```
function: compiling the following code:
*************************************
  (arg: Double) => {
    val x0: Double = arg * arg
    val x1: Double = x0 * x0
    val x2: Double = arg * x1
    x2: Double
  }
*************************************
3.0^5 = 243.0
4.0^5 = 1024.0
```

The generated code shows the `if` conditional and the recursive calls were eliminated. Indeed, running `pow` for the exponent `5` executes exactly three non-trivial operations transitively involving the argument `arg`, all three appearing in the generated code. This shows the operations were lifted and recorded in the operation graph, allowing the code above to be generated in the next stage. Let us see how the `pow` code was transformed to allow lifting.

In the case of staging, there is **no INJECT phase**, since the programmer manually marks the arguments to be `@staged`.

The COERCE phase follows the usual pattern of introducing coercions, with an additional constraint: immediate values can be coerced to staged constants, but not the other way around. This is done so that staging and compiling are only triggered explicitly, through calls such as `compile` and `function`. This restriction could easily be removed, but keeping it makes the performance predictable, as it puts the programmer in control of the lengthy staging and compilation process. Seen in relation to primitive types, when staging, boxing is cheap, but unboxing can potentially be expensive, so we want to trigger it explicitly.

The COERCE phase is also responsible for redirecting method calls for `@staged` receivers, which is essentially the lifting mechanism. Unlike the previous transformations, where extension methods were either provided by the library or extracted automatically, in the case of staging, they are manually written by the programmers. These methods are called infix methods [45] and they contain the mechanism to build the operation graph used to generate optimized code. Since this part is very similar to what is done in the LMS framework and is not our contribution, we point the reader to the works of *Rompf* [54, 55, 57] for more details.

The COMMIT phase transforms `@staged T` to the operation graph representation used in LMS, `Rep[T]`, and redirects calls to `compile` and `function` to `compile_impl` and `function_impl`, which trigger the synthesis and compilation for the operation graph.

The staging prototype serves to show that lifting language constructs can be modelled as an LDL-based representation transformation.

6.4.1 Evaluation

Lines of code and complexity. The staging plugin consists of 12 Scala files with 487 lines of code. The difference between the standard Scala compiler and Scala-virtualized is +2247/-578 LOC, including the library changes necessary to support lifting language constructs. Although the staging plugin is still far from being on-par with scala-virtualized in terms of lifting capabilities, it is 4 times smaller, despite the boilerplate necessary to create a Scala compiler plugin.

The COMMIT phase for the staging plugin has 110 lines of code and 5 transformation rules:

- 3 rules for redirecting markers to actual coercions;
- 2 rules for the special methods `compile` and `function`.

Runtime performance. We tested the staging plugin on the FFT example from Rosetta Code [6]. To stage the FFT example, we lifted the operations on complex numbers but left everything else to evaluate during staging. The separation into even and odd numbers and all the butterfly connections specific to FFTs are done only once during staging. Of course, this requires deciding on the number of elements ahead of time, thus fixing the batch size for the FFT analysis. With this, we get the following results:

```
::Benchmark FFT.Scala Complex::
Parameters(data size = 2^ -> 3): 0.966099

::Benchmark FFT.Stagium Complex::
Parameters(data size = 2^ -> 3): 0.018612
```

The times for executing the FFT (expressed in milliseconds) suggest that lifting the code and removing collection-related abstraction can bring a speedup of 53x, making staging worth it when running the FFT code multiple times.

The two extra features in the staging plugin are: (1) using programmer-written infix methods instead of synthetic or library extension methods and (2) the ability to restrict a class of coercions, in this case from staged to direct values, outputting meaningful error messages and explaining the problem to the user.

7. Related Work

Generics. Interoperation with generics motivates many of the data representation transformations in use today. The implementation of generics is influenced by two distinct choices: the choice of low-level code translation and the runtime type information stored.

The low-level code generated for generics can be either heterogeneous, meaning different copies of the code exist for different incoming argument types or homogeneous, meaning a single copy of the code handles all incoming argument types. Heterogeneous translations include Scala specialization [21], compile-time C++ template expansion [66] and load-time template instantiation [35] as done by the .NET CLR [9]. Homogeneous translations, on the other hand, require a uniform data representation, which may be either boxed values [15, 42], fixnums [77] or tagged unions [46].

In order to perform tests such as checking if a value is a list of integers at runtime, the type parameter must be taken into account. In homogeneous and load-time template expansions, one has to carry reified types for the type parameters. While this has an associated runtime cost [62], several solutions have been proposed to reduce it: in the CLR, reified types are computed lazily [35]. In Java, several papers presented viable schemes for carrying reified types, including PolyJ [11], Pizza [47], NextGen [19] and the work by *Viroli et al.* [76]. Finally, in ML, generic code (also called parametrically polymorphic in functional languages) can carry explicit type representations [30, 71].

Unboxed primitive types. In the area of unboxed primitive types, *Leroy* [42] presents a formal data representation transformation for the ML programming language based on typing derivations. The comparison in the introduction states that Late Data Layout introduces selectivity, object-oriented support and disentangles the transformation from its assumptions. This is a somewhat shallow comparison. A deeper comparison is that in *Leroy*'s transformation the INJECT and COMMIT phases are implicit and hard-coded while the two versions of the transformation rules presented by *Leroy* correspond to duplicating the COERCE phase for boxed and unboxed expected types. Instead of expected types, the ML transformation knows where generic parameters occur, and uses this information to invoke the correct version of the transformation. Therefore our main contribution is discovering and formulating the underlying principle and successfully extending it to a more broad context, to include value classes, specialization and staging, which have very different requirements.

Shao further extends *Leroy*'s work [63, 64] by presenting a more efficient representation, at the expense of carrying explicit type representations [30, 71]. *Minamide* further refines the transformation and is able to formally prove that the transformed code has the same time complexity as the original program [44]. Tracking value representation in types has been presented and extended to continuation-passing style [23] by *Thiemann* in [72]. Two pieces of information are tracked in a lattice: whether the value corresponding to the type is used at all (otherwise its representation can be ignored - called “Don't care polymorphism” and equivalent to our `oblivious` relation between AST nodes) and whether a certain representation is required. This information is used in a type inference algorithm which can elide coercions when the parameters are discarded or when a method call is in tail position, namely it doesn't need to box the result only to have the caller unbox it. It should be noted that the coercions operate on a continuation-passing-style intermediary representation.

A different direction in unboxing primitive types is based on escape analysis [20], where the program is analyzed at runtime and a local and conservative data representation transformation is performed. When implemented in just-in-time compilers [65] of virtual machines such as PyPy [13], Graal [79] or HotSpot [50], and coupled with aggressive inlining, the escape analysis can make an important difference, although it is limited by not being able to optimize containers outside its local scope. Late Data Layout and escape analysis are fundamentally different – escape analysis has a local scope and relies heavily on inlining, while LDL can safely optimize across method boundaries as long as the transformation consistently makes the same decisions in subsequent separate compilations. Interpreter-based techniques such as quickening [17] and trace-based specialization [24] can further improve escape analysis based on the dynamic execution profiles. Truffle [78] partially evaluates the interpreter for the running program and makes aggressive assumptions about the data representation, yielding the best results in terms of top speed at the expense of a longer warm-up time.

The Haskell programming language has two reasons to box primitive types in the low level code: (1) due to the non-strictness of the language, arguments to a function may not have been evaluated yet and are thus represented as thunks and (2) due to erased parametric polymorphism. Haskell exposes both the boxed `Int` representation and the unboxed `Int#`, although the compiler does transform `Int` values to `Int#` where possible. To do so, the Glasgow Haskell Compiler uses a syntax-based transformation coupled with a peephole optimization [32, 40]. In general, peephole optimizations have been formalized by *Henglein* in [31]. Haskell also features calling convention optimizations that make the argument laziness explicit and can unbox primitives in certain situations [12].

Value classes have been proposed for Java as early as 1999 [29, 58, 59]. The most recent description, which is also closest to our current approach, is the value class proposal for the Scala programming language [7]. We build upon the idea that a single concept should be exposed despite having multiple representations, but we step away from ad-hoc encodings and fixed rules in the type system. In this way, we can capture other representations, such as the tagged representation in [46]. Value classes have also been implemented in the CLR [2], but to the best of our knowledge the implementation has not been described in an academic setting. The Haskell programming language offers the `newtype` declaration [3] that, modulo the bottom type $\perp$, is unboxed similarly to value classes.

Specialization for generics is a technique aimed at eliminating boxing deep inside generic classes. Specialization has been implemented in Scala [21, 22] and has been improved by miniboxing [5, 75]. Specialization and macros have been combined to produce a mechanism for ad-hoc specialization of code in Scala [67]. The .NET CLR automatically specializes all generics, thanks to its bytecode metadata and reified types [35].

A different approach to deep boxing elimination is described for Haskell [33] and Python [14]. It relies on specializing arrays while providing generic wrappers around them. This allows memory-efficient storage without the complex problem of providing heterogeneous translations for each of the methods exposed by data structures.

Multi-stage programming (also called staging) [70] requires lifting certain expressions in the program to a reified representation. Staging can be implemented using macros [18, 25, 34], or using specialized compiler extensions [45].

One of the applications is removing the abstraction overhead of high-level and embedded domain specific languages. Indeed, staging was successfully used to optimize and re-target domain-specific languages (DSLs) [16, 37, 54, 55, 57, 74].

Annotated types [4, 8] have been introduced to trigger code transformations and to allow the extension of the type system into the area of program verification while reusing as much infrastructure from the compiler as possible [51]. In the context of Java, type annotations have been used to selectively add reified type argument information to erased generics [27]. In the context of Scala, annotated types have been used to track and limit the side-effects of expressions [60, 61], to designate macro expansions [18] and to trigger continuation-passing-style transformations [56].

Formalization. In [42], *Leroy* presents a full formalization for the primitive unboxing for ML, including a proof of operational equivalence. The .NET generics are formalized in [80]. An effort to formalize LDL is currently on-going [73] and it relies on local type inference, as described by *Odersky et al.* [48] and *Pierce et al.* [52].

In the area of formal descriptions, two papers on type-directed coercion insertion stand out as very closely related to this paper [41, 69]. The work of *Swami* et al. [69] focuses on automatically composing several coercions together in order to bridge the gap between different types. The highlight of the paper are the powerful composition rules and the proofs that, despite their generality, always produce syntactically unique, non-ambiguous rewritings. This work resembles the mechanisms used to introduce implicit conversions in Scala, although the rules provide more flexibility and are proven not to diverge. On the other hand, *Leather* et al. [41] describe a coercion insertion mechanism which deliberately produces ambiguous rewritings from which heuristics can pick the best. More importantly, the formalism presented in [41] is also capable of consistently changing types in the rewrite rules, making the transformation very versatile. Unfortunately, the two formalisms do not handle backward propagation, object orientation and subtyping, all of which are crucial to performing optimal data representation transformations in Scala. Furthermore, they do not provide the ability to selectively transform the data representation, making them unusable for the three use cases we presented. By comparison, an important limitation of our work is that the `box` and `unbox` coercions we introduce are un-ambiguous and not composible by design, as we aim for a one-step conversion between different representations.

8. Acknowledgements

We would like to thank Aymeric Genêt, who developed the least squares benchmark for the miniboxing plugin [26].

We are grateful to the Scala teams at EPFL and Typesafe for providing precious feedback and helping shape the representation mechanism we have today. In particular, we would like to thank Manohar Jonnalagedda, Dmitry Petrashko, Iulian Dragos, Miguel Garcia and Lukas Rytz for the discussions we had, which always led to interesting developments. We are thankful to the Scala community, for trying the project, reporting bugs and providing cool new ideas. We would like to thank our paper and artifact reviewers in the OOPSLA conference for providing clear and concise feedback, which guided us in improving the paper.

Last but not least, Vlad is grateful to his wife Ana Lucia and his family who supported him through very difficult times when this paper was written.

9. Conclusion

In this paper we presented a general mechanism that allows refining a high-level concept into multiple representations. This is done in a selective way, by annotating values in the program with their desired representation. The coercions necessary for maintaining program consistency with regards to representations are introduced automatically, consistently and optimally thanks to local type inference.

We validated the algorithm for three cases: multi-parameter value classes, specialization through miniboxing and a simple multi-stage programming mechanism. The results were encouraging: we were able to reuse much of the infrastructure (which has been developed as part of the miniboxing plugin) for the other plugins and the development time was in the order of developer-weeks.

Finally, the key insights of the paper are that annotated types are a perfect vehicle for carrying representation information and introducing coercions can be done consistently and optimally using the expected type mechanism in local type inference.

References

[1] Count lines of code. URL `http://cloc.sourceforge.net/`.

[2] Value Types in the Common Type System, Microsoft Developer Network. URL `http://msdn.microsoft.com/en-us/library/34yytbws.aspx`.

[3] Haskell 98 Language and Libraries: Section 4.2.3. URL `http://www.haskell.org/onlinereport/decls.html#sect4.2.3`.

[4] JSR 308: Annotations on Java Types. URL `https://jcp.org/en/jsr/detail?id=308`.

[5] The Miniboxing plugin website. URL `http://scala-miniboxing.org`.

[6] Rosetta Code Website. URL `http://rosettacode.org`.

[7] Scala SIP-15: Value Classes. URL `http://docs.scala-lang.org/sips/completed/value-classes.html`.

[8] SIP-5 - Internals of Scala Annotations. URL `http://docs.scala-lang.org/sips/completed/internals-of-scala-annotations.html`.

[9] ECMA International, Standard ECMA-335: Common Language Infrastructure, June 2006.

[10] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded DSL for high performance big data processing. In *Big Data*, 2012.

[11] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized Types for Java. In *PoPL*. ACM, 1997.

[12] M. C. Bolingbroke and S. L. Peyton Jones. Types Are Calling Conventions. In *Haskell*. ACM, 2009.

[13] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *ICOOLPS*. ACM, 2009.

[14] C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA*. ACM, 2013.

[15] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *OOPSLA*. ACM, 1998.

[16] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*. IEEE Computer Society, 2011.

[17] S. Brunthaler. Efficient Interpretation Using Quickening. In *DLS*. ACM, 2010. .

[18] E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *SCALA*. ACM, 2013.

[19] R. Cartwright and G. L. Steele, Jr. Compatible Genericity with Run-time Types for the Java Programming Language. In *OOPSLA*. ACM, 1998.

[20] A. Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-order Functional Specifications. In *POPL*. ACM, 1990. .

[21] I. Dragos. *Compiling Scala for Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2010.

[22] I. Dragos and M. Odersky. Compiling Generics Through User-Directed Type Specialization. In *ICOOOLPS*, Genova, Italy, 2009.

[23] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *PLDI*. ACM, 1993.

[24] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *PLDI*. ACM, 2009.

[25] S. E. Ganz, A. Sabry, and W. Taha. Macros As Multi-stage Computations: Type-safe, Generative, Binding Macros in MacroML. In *ICFP*. ACM, 2001.

[26] A. Genêt, V. Ureche, and M. Odersky. Improving the Performance of Scala Collections with Miniboxing (EPFL-REPORT-200245). Technical report, EPFL, 2014. URL `http://scala-miniboxing.org/`.

[27] P. Gerakios, A. Biboudis, and Y. Smaragdakis. Reified Type Parameters Using Java Annotations. In *GPCE*. ACM, 2013.

[28] B. Goetz. The State of Speclaization, 2014. URL `http://web.archive.org/web/20140718191952/http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html`.

[29] J. Gosling. The Evolution of Numerical Computing in Java - preliminary discussion on value classes. URL `http://web.archive.org/web/19990202050412/http://java.sun.com/people/jag/FP.html#classes`.

[30] R. Harper and G. Morrisett. Compiling Polymorphism Using Intensional Type Analysis. In *PoPL*. ACM, 1995.

[31] F. Henglein and J. Jørgensen. Formally Optimal Boxing. In *PoPL*. ACM, 1994.

[32] S. L. P. Jones and J. Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture*. Springer, 1991.

[33] S. L. P. Jones, R. Leshchinskiy, G. Keller, and M. M. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*, volume 2, pages 383–414, 2008.

[34] V. Jovanovic, A. Shaikhha, S. Stucki, V. Nikolaev, C. Koch, and M. Odersky. Yin-Yang: Concealing the Deep Embedding of DSLs. 2014.

[35] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *PLDI*, 2001.

[36] G. A. Kildall. A unified approach to global program optimization. In *PoPL*. ACM, 1973.

[37] G. Kossakowski, N. Amin, T. Rompf, and M. Odersky. JavaScript As an Embedded DSL. In *ECOOP*. Springer, 2012.

[38] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot Client Compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1), 2008.

[39] P. A. Kulkarni. JIT Compilation Policy for Modern Machines. In *OOPSLA*. ACM, 2011.

[40] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In *ESOP*. Springer, 1996.

[41] S. Leather, J. Jeuring, A. Löh, and B. Schuur. Type-changing Rewriting and Semantics-preserving Transformation. In *PEPM '14*. ACM, 2014.

[42] X. Leroy. Unboxed Objects and Polymorphic Typing. In *PoPL*. ACM, 1992.

[43] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[44] Y. Minamide and J. Garriguc. On the Runtime Complexity of Type-Directed Unboxing. In *ICFP*, 1998.

[45] A. Moors, T. Rompf, P. Haller, and M. Odersky. Scala-Virtualized. In *PEPM*. ACM, 2012.

[46] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An Ad Hoc Approach to the Implementation of Polymorphism. *ACM TOPLAS*, 1991.

[47] M. Odersky, E. Runne, and P. Wadler. *Two Ways to Bake Your Pizza-Translating Parameterised Types into Java*. Springer, 2000.

[48] M. Odersky, M. Zenger, and C. Zenger. Colored Local Type Inference. In *PoPL*. ACM, 2001.

[49] E. Osheim. Generic Numeric Programming Through Specialized Type Classes. *ScalaDays*, 2012.

[50] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 2001.

[51] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *ISSTA*. ACM, 2008.

[52] B. C. Pierce and D. N. Turner. Local Type Inference. *ACM TOPLAS*, 2000.

[53] A. Prokopec. ScalaMeter. URL `http://axel22.github.com/scalameter/`.

[54] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2012.

[55] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*, 2010. .

[56] T. Rompf, I. Maier, and M. Odersky. Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform. In *ICFP*. ACM, 2009.

[57] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-Blocks for Performance Oriented DSLs. In *DSL*, 2011.

[58] J. Rose. Value Types and Struct Tearing , . URL `https://web.archive.org/web/20140320141639/https://blogs.oracle.com/jrose/entry/value_types_and_struct_tearing`.

[59] J. Rose. Value Types in the VM, . URL `http://web.archive.org/web/20131229122932/https://blogs.oracle.com/jrose/entry/value_types_in_the_vm`.

[60] L. Rytz. *A Practical Effect System for Scala*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2014.

[61] L. Rytz, M. Odersky, and P. Haller. Lightweight Polymorphic Effects. In *ECOOP*. Springer, 2012.

[62] M. Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

[63] Z. Shao. Flexible Representation Analysis. In *ICFP*. ACM, 1997.

[64] Z. Shao and A. W. Appel. A Type-Based Compiler for Standard ML. In *PLDI*, 1995.

[65] L. Stadler, T. Würthinger, and H. Mössenböck. Partial Escape Analysis and Scalar Replacement for Java. In *CGO*. ACM, 2014.

[66] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 3rd edition, 1997.

[67] N. Stucki and V. Ureche. Bridging islands of specialized code using macros and reified types. In *SCALA*. ACM, 2013.

[68] S. Stucki, N. Amin, M. Jonnalagedda, and T. Rompf. What Are the Odds?: Probabilistic Programming in Scala. In *SCALA*. ACM, 2013.

[69] N. Swamy, M. Hicks, and G. M. Bierman. A Theory of Typed Coercions and Its Applications. In *ICFP '09*. ACM, 2009.

[70] W. Taha. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[71] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *PLDI*. ACM, 1996.

[72] P. J. Thiemann. Unboxed Values and Polymorphic Typing Revisited. In *Functional Programming Languages and Computer Architecture*. ACM, 1995.

[73] V. Ureche. Additional Material for "Unifying Data Representation Transformations (EPFL-REPORT-200246)". Technical report, EPFL, 2014.

[74] V. Ureche, T. Rompf, A. Sujeeth, H. Chafi, and M. Odersky. StagedSAC: A Case Study in Performance-oriented DSL Development. In *PEPM*. ACM, 2012.

[75] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *OOPSLA*, 2013.

[76] M. Viroli and A. Natali. Parametric Polymorphism in Java: An Approach to Translation Based on Reflective Features. In *OOPSLA*. ACM, 2000.

[77] S. Wholey and S. E. Fahlman. The Design of an Instruction Set for Common Lisp. In *LFP*, 1984.

[78] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST interpreters. In *DLS*. ACM, 2012.

[79] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Onward!* ACM, 2013.

[80] D. Yu, A. Kennedy, and D. Syme. Formalization of Generics for the .NET Common Language Runtime. In *POPL*, POPL '04. ACM.