Automating Ad hoc Data Representation Transformations

Abstract
To maximize the run-time performance, programmers specialize their code by hand, replacing library collections and containers by custom objects in which the data is restructured for efficient access. However, this is a tedious and error-prone process that makes it hard to test, maintain and evolve the source code.

We present an automated and composable approach that allows programmers to safely change the data representation in delimited scopes containing anything from expressions to entire class definitions. The transformation itself is defined by programmers and can cover a wide range of use cases.

Our technique leverages the type system in order to infer where the data representation needs to be converted, while offering a strong correctness guarantee on the interaction with other object-oriented language features, such as dynamic dispatch, inheritance and generics.

We have embedded this technique in a Scala compiler plugin and used it in four very different transformations, ranging from improving the data layout and encoding, to retrofitting specialization and value class status, and to collection deforestation. These transformations obtain speedups between 1.9 and 13x on user programs.

Keywords  
data representation, jvm, bytecode, compatibility, transformation, optimization, safety, semantics

1. Introduction
An object encapsulates code and data and exposes an interface. Modern language facilities, such as extension methods, type classes and implicit conversions allow programmers to evolve the object interface in an ad hoc way, by adding new methods and operators. For example, in Scala, we can use an implicit conversion to add the multiplication operator to pairs of integers, with the semantics of complex number multiplication:

```
scalac> (0, 1) * (0, 1)
res0: (Int, Int) = (-1, 0)
```

Unlike evolving the interface, no general mechanism in modern languages is capable of evolving an object’s encapsulated data. The encapsulated data is assumed to be fixed, allowing the compiled code to contain hard references to data, encoded according to a convention known as the object layout. For instance, methods encapsulated by the generic pair class, such as `swap` and `toString`, rely on the existence of two generic fields, erased to `Object`. This leads to inefficient storage in the running example, as the integers need to be boxed, producing as many as 3 heap objects for each “complex number”: the two boxed integers and the pair container. What if, for a part of our program, instead of the pair, we concatenated the two 32-bit integers into a 64-bit long integer, that would represent the “complex number”? We could pass complex numbers by value, completely sidestepping the need to allocate memory and to garbage-collect it later. Additionally, what if we could also add functionality, such as arithmetic operations, to our ad hoc complex numbers, all without any heap allocation overhead? Finally, what parts of such a transformation could be automated?

Object layout transformations are common in dynamic language virtual machines, such as V8 and Truffle. These virtual machines profile values at run-time and make optimistic assumptions about the shape of objects. This allows them to automatically optimize the object layout in the heap, at the cost of recompiling of all the code that references the old object layout. If, later in the execution, the assumptions prove too optimistic, the virtual machine needs to revert to the more general (and less efficient) object layout, again recompiling all the code that contains hard references to the optimized layout. As expected, this comes with important overheads. Thus, runtime decisions to change the low-level object layout are both expensive (due to recompilation) and have a global nature, affecting all code that assumes a certain layout.

Since transforming the object layout at run-time is expensive, a natural question to ask is whether we can leverage the statically-typed nature of a programming language to optimize the object layout during compilation? The answer is yes. Transformations such as “class specialization” and “value class inlining” transform the object layout in order to avoid the creation of heap objects. However, both of these transformations take a global approach: when a class is marked as specialized or as a value class (and assuming it satisfies the semantic restrictions) it is transformed at its definition site. Later on, this allows all references to the class, even in separately compiled sources, to be optimized. On the other hand, if a class is not marked at its definition site, retrofitting specialization or the value class status is impossible, as it would break many non-orthogonal language features, such as dynamic dispatch, inheritance and generics.

Therefore, although transformations in statically typed languages can optimize the object layout, they do not meet

[Copyright notice will appear here once 'preprint' option is removed.]
the ad hoc criterion: they cannot be retrofitted later, and they
have a global, all-or-nothing nature. For instance, in Scala,
the generic pair class is specialized but not marked as a value
class. As a result, the representation is not fully optimized,
still requiring a heap object per pair. Even worse, specializa-
tion and value class inlining are mutually exclusive, making
it impossible to optimally represent our “complex numbers”
as values even if we had complete control over the Scala li-
brary. Furthermore, a change in the data representation may
be applicable for specific parts of the client code, but might
not make sense to apply globally.

In our “complex numbers” abstraction, we only use a frac-
tion of the flexibility provided by the library tuples, and yet we have to give up all the code optimality. Even
worse, for our limited domain, we are aware of a better
representation, but the only solution is to transform the code
by hand, essentially having to choose between an obfuscated
or a slow version of the code. What is missing is a largely
automated and safe transformation that allows us to use
our domain-specific knowledge to mark a scope where the
“complex numbers” can use the alternative object layout,
effectively specializing that part of our program.

In this paper we present such an automated transfor-
mation that allows programmers to safely change the data
representation in limited, well-defined scopes that can in-
clude anything from expressions to method and class defini-
tions. The transformation maintains strong correctness guar-
antees in terms of non-orthogonal language features, such as
dynamic dispatch, inheritance and generics across separate
compilations. To gain the most benefit, our approach uses
the programmers domain-specific knowledge of the transformed
scope, allowing them to specify the exact alternative repre-
sentation and the operations it should expose, while com-
pletely automating all the tedium involved in safely trans-
forming the code and maintaining the outside interface.

This way, the programmer is responsible for correctly
stating (a) what the data representation transformation is
and (b) to which program scope it applies. Our approach is
then responsible for (1) automatically deciding when to ap-
ply the transformation and when to revert it, in order to en-
sure correct interchange between representations, (2) enrich-
ing the transformation with automatically generated bridge
code that ensures correctness relative to overriding and dy-
namic dispatch and (3) persisting the necessary metadata to
allow transformed program scopes in different source files
and compilation runs to communicate using the optimized
representation – a property we refer to as composable in the
following sections. Thus, our approach adheres to the de-
sign principle of separating the reusable, general and prov-
ably correct mechanism from the programmer-defined pol-
icy, which may contain incorrect decisions [26].

Our main contributions are:
• Introducing the ad hoc data representation problem,
which, to the best of our knowledge, has not been ad-
dressed at all in the literature (§2);
• Presenting the extensions that allow global data rep-
resentation transformations (§3) to be used as scoped
programmer-driven transformations (§4);
• Implementing the approach presented as a Scala compiler
plugin [2] that allows programmers to express custom
transformations (§5) and benchmarking the plugin on a
broad spectrum of transformations, ranging from impro-
vining the data layout and encoding, to retrofitting special-
ization and value class status, and to collection deforesta-
tion [42]. These transformations produced speedups be-
tween 1.9 and 13x on user programs (§6).

2. Motivation

This section presents the full motivating example featuring
the complex number transformation, which we use through-
out the paper. It then shows how the data representation
transformation is triggered and introduces the terminology.
Finally, it shows a naive transformation, hinting at the diffi-
culties lying ahead.

2.1 Motivating Example

In the introduction, we focused on adding complex number
semantics to pairs of integers. Complex numbers with in-
tegers as both their real and imaginary parts are known as
Gaussian integers, and are a countable subset of all complex
numbers. The operations defined on Gaussian integers are
similar to complex number operations, with one exception:
to satisfy the abelian closure property, division is not precise,
but instead rounds the result to the nearest Gaussian integer,
with both the real and imaginary axes containing integers.
This is similar to integer division, which also rounds the re-
sult, so that, for example, 5/2 produces value 2.

An interesting property of Gaussian integers is that we
can define the “divides” relation and the greatest common
divisor (GCD) between any two Gaussian integers. Further-
more, computing the GCD is similar to Euclid’s algorithm
for integer numbers:

```scala
def gcd(n1: (Int, Int), n2: (Int, Int)): (Int, Int) = {
  val remainder = n1 % n2
  if (remainder.norm == 0) n2 else gcd(n2, remainder)
}
```

Unfortunately, as our algorithm recursively computes the
result, it creates linearly many pairs of integers, allocating
them in the heap memory. If we run this algorithm with the
heap allocations and no optimizations, computing the GCD
takes around 3 microseconds (on the same setup as used for
our full experiments in §6):

```scala
scala> timed() => gcd((544, 185), (131, 181))
The operation took 3.05 us (based on 10000 executions).
The result was (10, 3).
```

However, if we encode the Gaussian integers into 64-bit
long integers, we improve the time by a factor of 13x:

```scala
scala> timed() => gcd((544, 185), (131, 181))
The operation took 0.23 us (based on 10000 executions).
The result was (10, 3).
```
This makes the data representation transformation highly desirable. Still, making the programmer transform the code by hand is tedious and error-prone, so a natural question to ask is whether the transformation could be automated?

2.2 Automating the Transformation

In order to reap the benefits of using the improved representation without manually transforming the code, we present the Ad hoc Data Representation (ADR) Transformation technique, which can be triggered by the adrt marker method. This method accepts two parameters: the first parameter is a transformation description object and the second is a block of code that forms the transformation scope. This scope can contain anything from expressions all the way to method, class, trait and object definitions:

```scala
adrt[IntPairComplexToLongComplex] {
  def gcd(n1: (Int, Int), n2: (Int, Int)): (Int, Int) = {
    val remainder = n1 % n2
    if (remainder.norm == 0) n2 else gcd(n2, remainder)
  }
}
```

To maintain a consistent naming throughout the paper, we will use the name high-level type to designate (Int, Int), which corresponds to the original type in the code. This high-level type can be encoded as its representation type, Long. The high-level type, its representation, and the procedures for encoding and decoding are all stored in the transformation description object, in our case IntPairComplexToLongComplex. With this, we have the vocabulary necessary to reason about our first (naive) approach to transforming the code.

2.3 A Naive Transformation

Despite its simple interface, the Ad hoc Data Representation Transformation mechanism is by no means simple. For example, knowing that the adrt marker was instructed to transform (Int, Int) to Long, a naive result could be:

```scala
def gcd(n1: Long, n2: Long): Long = {
  val remainder = n1 % n2
  if (remainder.norm == 0) n2 else gcd(n2, remainder)
}
```

There are many questions one could ask about this naive translation. For example, how does the compiler know which parameters and values to transform to the long integer representation? How and when to encode and decode values, and what to do about values that are visible outside the scope? Even worse, what if parts of the code are compiled separately, in a different compiler run?

Going into the semantics of the program, we can ask if the % (modulo) operator maintains the semantics of complex numbers when used for long integers? Also, is norm defined for long integers? Unfortunately, the response to both questions is negative. Therefore, the transformation needs to preserve semantics, which is not trivial.

We could also ask what would happen if gcd was overriding another method? Would the new signature still override it? The answer is no, so the naive translation would break the object model.

Our approach, the Ad hoc Data Representation Transformation, answers all these questions and preserves semantics.

3. Data Representation Transformations

As necessary background for our approach, we review data representation transformations in general and, in particular, the Late Data Layout (LDL) transformation mechanism [41], which we later extend to our ad hoc data representation transformation (§4). The LDL mechanism is neither programmer-driven, since the data representation has to be known a priori and encoded in the transformation, nor directly applicable to limited scopes inside a program.

Data can usually be represented in several ways, some more efficient and others more flexible. For example, integer numbers can use either the primitive (unboxed) value encoding, which is more efficient, or the object-based (boxed) encoding, which is more flexible. The boxed representation allows integers to act as the receivers of dynamically dispatched method calls, to be assigned to supertypes, such as Number or Object and to instantiate erased generics. However, the extra flexibility comes at a price: boxed integers are allocated on the heap so they need to be garbage-collected later and all their operations incur an indirection overhead. This leads to a tension between the two representations.

From a language perspective, there are two approaches to exposing the multiple representations of a type: either have a different type for each representation, as Java does, or fully hide the difference and present a single language-level type, as ML, Haskell and Scala do. Either way, the final low-level bytecode or assembly code needs to handle the two representations separately, since they correspond to very different entities: references and values.

Exposing a single high-level type in the language is more popular among programmers for its simplicity, but it places more responsibility on the compiler, which has to perform two additional steps: first, it needs to choose the data representation of each value; and second, it needs to introduce coercions that switch between representations where necessary. For example, since only boxed integers can instantiate generics, any unboxed integer going into a generic container, such as a list, needs to be coerced to the boxed representation. This work is done in the compiler pipeline, in so-called data representation transformations.

The Late Data Layout mechanism, presented next, is a powerful data representation transformation facility for Scala. It has three properties that make it well-suited to be a substrate for our Ad hoc Data Representation Transformation: selectivity, optimality and consistency.
3.1 Late Data Layout

The Late Data Layout (LDL) mechanism [41] is the underlying transformation used in Scala to implement multi-parameter value class inlining and to specialize classes using the miniboxed encoding [40]. It is a flexible and reliable mechanism, tested on many thousands of lines of code.

Using LDL, a language can expose high-level types (called high-level concepts in the LDL terminology), such as the integer type `Int` exposed by Scala, which can represent either a boxed or unboxed value in the low-level bytecode. In the following example, we have values of types `Int` and `Any`. `Any` is the top of the Scala type system, and thus a supertype of `Int`:

```scala
val i: Int = 1
val j: Int = 1
val k: Any = j
```

Since Scala compiles down to Java bytecode, during compilation, the LDL-based primitive unboxing transformation bridges the gap between the high-level `Int` concept and its two representations: the unboxed `int` and the boxed `java.lang.Integer` representation. Along the way, it introduces the necessary coercions between these two representations. For example, the code above is translated to:

```scala
l1: @unboxed Int = 1 // expected/found: @unboxed
l2: @unboxed Int = 1 // expected/found: @unboxed
```

The LDL mechanism transforms the data representation in three phases: INJECT, COERCSE and COMMIT. Each of the phases is responsible for a property of the transformation: INJECT makes LDL selective. COERCSE makes it optimal and COMMIT makes it consistent. In our examples, we show the equivalent source code for the program abstract syntax trees (ASTs) after each of these phases.

The **INJECT phase** is responsible for marking each value with its desired representation. In the case of primitive integer unboxing, the annotation is `@unboxed`, and it signals that a value should be stored in the unboxed `int` representation. As an optimization, instead of adding a `@boxed` annotation for the corresponding cases, values that are not marked are automatically considered boxed. Following the INJECT phase, the previous example will be transformed to:

```scala
val i: @unboxed Int = 1 // Int can be unboxed
val j: @unboxed Int = 1 // Int can be unboxed
val k: Any = j // Any cannot be unboxed
```

The INJECT phase gives LDL a selective nature, allowing it to mark each individual value with its representation. For example, it would have been equally correct if the marking rules decided that `j` should be boxed, in which case its type would not have been marked. One of the properties of the LDL transformation is that boxed and unboxed values are compatible in the INJECT phase, so there are no coercions.

The **COERCSE phase**, as its name suggests, introduces coercions. This is done by changing the annotation semantics: annotated types become incompatible among themselves and with their un-annotated counterparts. This change in the annotation semantics corresponds to introducing the different representations: each annotation corresponds to a representation, and representations are not compatible with each other. With this change, an assignment from one representation to another will lead to mismatching types. Therefore, by re-type-checking the tree, the COERCSE phase can detect representation mismatches and can patch them using coercions.

In the example, the last line contains such a mismatch:

```scala
val i: @unboxed Int = 1 // expected/found: @unboxed
val j: @unboxed Int = 1 // expected/found: @unboxed
val k: Any = box(j) // mismatch => box
```

The COERCSE phase establishes the optimality property of the LDL transformation. The definition of optimality is quite involved, but we can easily show it using an example. Consider the following two integer definitions:

```scala
val c: Boolean = ...
val l1: @unboxed Int = if (c) i else j
val l2: @unboxed Int = unbox(if (c) box(i) else box(j))
```

It is clear that the two definitions will always produce the same result. Yet, the first one is markedly better: it does not execute any coercions, compared to second definition, which executes two coercions regardless of the value of `c`. These subtle sub-optimalities can slow down program execution, increase the heap footprint and the bytecode size. This is why the COERCSE phase needs to be optimal. The LDL paper [41] makes the following intuition-based conjecture: “in any given terminating execution trace through the transformed program, the number of coercions executed is minimal, for given sets of annotations introduced by the INJECT phase and transformations performed in the COMMIT phase”. An initial formalization and proof is sketched in [39].

From our perspective, optimality means that once representations are chosen and annotated, only the necessary coercions will be introduced during the COERCSE phase.

The **COMMIT phase** is responsible for introducing the actual representations. In the case of primitive unboxing, `@unboxed Int` is replaced by `int`, and `Int`, which is considered boxed, is replaced by `java.lang.Integer`. The box and unbox coercions are also replaced by the creation of objects and, respectively, by the extraction of the unchecked value:

```scala
val i: int = 1
val j: int = 1
val k: Integer.valueOf() =
```

The COMMIT phase is responsible for the consistency of the transformation. Since the program abstract syntax tree (AST) has been checked by the type-system extended with representation semantics, the COMMIT phase is guaranteed to correctly handle the value representations and to correctly coerce between them. This allows the COMMIT phase to be a very simple, syntax-based, transformation over the program abstract syntax tree (AST).

3.2 Support For Object-Oriented Programming

The LDL mechanism is aimed at object-oriented programming languages, which pose unique challenges for data representation transformations. This section will describe
the additional rules necessary in LDL to handle object-orientation.

**Object-oriented Patterns.** Aside from introducing coercions, data representation transformations must handle object-oriented patterns, such as method calls and subtyping. Not all representations can be used with these patterns. For example, it is not possible to call the `toString` method on the unboxed `Int` representation:

```
val a: @unboxed Int = 1
println(a.toString)
```

To handle dynamically dispatched method calls, LDL has a built-in rule: when a value acts as a method call receiver, it is coerced to the boxed representation, which, in this case, corresponds to the non-annotated representation. In our example, the `@unboxed Int` value is boxed during the `COERCES` phase, so it can act as the receiver of the `toString` method:

```
val a: @unboxed Int = 1
println(box(a).toString)
```

To improve performance, the LDL mechanism also supports bypass methods, also known as extension methods in the literature. For example, if a static `bypass_toString` method is available for the unboxed `Int` representation, there is no need to convert it before the method call:

```
val a: @unboxed Int = 1
println(bypass_toString(a))
```

Subtyping is handled in a similar fashion, by requiring the boxed representation, which can be assigned to supertypes.

**Support for Generics.** The Late Data Layout mechanism is agnostic to generics. This means that, depending on the transformation semantics and the implementation of generics, the mechanism can inject annotations in the type arguments or not. For example, if generics are erased, a list of integers will have type `List[Int]`, since values need to be boxed. If generics are unboxed and reified, the list type will be `List[unboxed Int]`. In the LDL paper [41], the authors show examples of both cases: when annotations are propagated inside generics and when they are not. The LDL mechanism adapts seamlessly to either case.

Having seen the Late Data Layout mechanism at work for unboxing primitive types, we can now look at how it can be extended to handle ad hoc programmer-driven data representation transformations.

### 4. Ad hoc Data Representation Transformation

The Ad hoc Data Representation (ADR) transformation adds two new elements over existing data representation transformations: (1) it enables custom, programmer-defined alternative representations and (2) it allows the transformation to take place in limited scopes, ranging from expressions all the way to method and class definitions. This allows programmers to use locally correct transformations that may be incorrect for code outside their given scope.

Section 2.2 showed how the ADR transformation is triggered by the `adrt` marker. The running example is reproduced below for quick reference:

```scala
object IntPairComplexToLongComplex {  
  def gcd(n1: (Int, Int), n2: (Int, Int)): (Int, Int)= {  
    val remainder = n1 % n2
    if (remainder.norm == 0) n2 else gcd(n2, remainder)  
  }  
}
```

The following sections take a step by step approach to explaining how our technique allows programmers to define transformations and to use them in localized program scopes, improving the performance of their programs in an automated and safe fashion.

#### 4.1 Transformation Description Objects

The first step in performing an `adrt` transformation is defining the transformation description object. This object is required to extend a marker interface and to define the transformation through the `toRepr` and `toHigh` coercions:

```scala
object IntPairComplexToLongComplex {  
  extends TransformationDescription {  
    // coercions:
    def toRepr(high: (Int, Int)): Long = ...  
    def toHigh(repr: Long): (Int, Int) = ...  
    // bypass methods:
    ...  
  }
}
```

The coercions serve a double purpose: (1) the signatures match the high-level type, in this case `(Int, Int)` and indicate its corresponding representation type, `Long` and vice-versa and (2) the implementations are called in the transformed scope to encode and decode values as necessary.

**Bypass Methods.** The description object can optionally include bypass methods, which correspond to the methods exposed by the high-level type but operate on encoded values in the representation type. Bypass methods allow the transformation to avoid coercing receivers to the high-level type by rewriting dynamically dispatched calls to their corresponding statically-resolved bypass method calls, as shown in section §3.2. Method call rewriting in `adrt` scopes is explained later, in section §4.4.

**Generic Transformations.** In our example, both the high-level and representation types are monomorphic (i.e. not generic). Still, in some cases, the `adrt` transformation is used to target library collections regardless of the type of their elements. Although we studied multiple approaches to defining the transformation description objects, we converged on the current approach, which has the merit of being concise and naturally extending to generic high-level and representation types, by allowing the coercions to be generic themselves:

```scala
def toRepr[T](high: List[T]): LazyList[T] = ...  
def toHigh[T](repr: LazyList[T]): List[T] = ...
```

Since the coercion signatures “match” the high-level type and return the corresponding representation type, a value of type `List[Int]` will be matched by the `adrt` transformation and subsequently encoded as a `LazyList[Int]`. This allows the `adrt` scopes to transform collections, containers...
and function representations. The benchmarks section (§6) shows two examples of generic transformations.

**Target Semantics.** It is worth noting that coercions defined in transformation objects must observe the semantics of the high-level type. In particular, semantics such as mutability and referential identity must be preserved if the program relies on them. For example, correctly handling referential identity requires the coercions to return the exact same object (the exact same reference) when interleaved:

```scala
assert(toHigh(toRepr(x)) eq x) // referential equality
```

These semantics reduce the benefit of the `adrt` transformation by imposing restrictions on the coercions. However, in most use cases, the targets, such as Scala collections and containers, have value semantics: they are immutable, final and only use structural equality. Such high-level types can be targeted at will, since they can be reconstructed at any time without the program observing it.

Once the transformation description object is defined, it can be used in `adrt` scopes to optimize the user program.

### 4.2 Transformation Scopes and Composability

Unlike existing data representation transformations, such as value class inlining and specialization, which have fixed semantics and occur in a sequence, the ADR transformation handles all transformation scopes in the source code concurrently, each with its own high-level target, representation type and coercions. This is a challenge, as handling the interactions between these concurrent scopes, some of which may even be nested, demands a highly disciplined treatment.

The key to handling all concurrent scopes correctly is shifting focus from the scopes themselves to the values they define. Since we are using the underlying LDL mechanism, we can track the encoding of each value in its type, using annotations. To keep track of the different transformations introduced by different scopes, we extend the LDL annotation to reference the description object, essentially carrying the entire transformation semantic with each individual value. We then leverage the type system and the signature persistence facilities to correctly transform all values, essentially allowing scopes to safely and efficiently pass values among themselves, using the representation type—a property we call composability.

We look at four instances of composability:

- allowing different scopes to communicate, despite using different representation types (high-level types coincide);
- isolating high-level types, barring unsound value leaks through the representation type;
- handling conflicting transformation description objects;
- passing values between high-level types in the encoded (representation) format;

Although the four examples cover the most interesting corner cases of the transformation, should the readers be interested, all cases of scope overlapping and nesting are described on the project wiki [3], on the “Scope Nesting” page. Furthermore, the scope composition is tested with each commit, as part of the project’s test suite.

**A high-level type can have different representations in different scopes.** This goes according to the scoped nature of the ADR transformation, which allows programmers to safely use the most efficient data representation for each task. Yet, this raises the question of whether values can be safely passed across scopes using different representations:

```scala
: adrt(IntPairToLong) { var x = (3, 5) }
: adrt(IntPairToDouble) { val y = (2, 6); x = y }
```

At a high level, the code is correct: the variable `x` is set to the value of `y`, both of them having high-level type `(Int, Int)`. However, being in different scopes, these two values will be encoded differently, `x` as a long integer and `y` as a double-precision floating point number. In this situation, how will the assignment `x = y` be translated? Let us look at the transformation step by step.

After parsing, the scope is inlined and the program is type-checked against the high-level types. Aside from checking the high-level types, the type checker also resolves implicits and infers all missing type annotations. During time, the description objects are kept as tree attachments:

```scala
var x: (Int, Int) = (3, 5)
val y: (Int, Int) = (2, 6)

x = y
```

Then, during the INJECT phase, for each defined value or method, the attached description objects are added to the `@repr` annotation, parameterized on the transformation description object:

```scala
var x: @repr(IntPairToLong) (Int, Int) = (3, 5)
val y: @repr(IntPairToDouble) (Int, Int) = (2, 6)

x = y
```

The `@repr` annotation is only attached if the value’s type matches the high-level type in the description object. Therefore, programmers are free to define values of any type in the scope, but only those values whose type matches the transformation description object’s target will be annotated.

Based on the annotated types, the COERC phase notices the mismatching transformation description objects in the last line: the left-hand side is on its way to be converted to a long integer (based on description object `IntPairToLong`) while the right-hand side will become a floating point expression (based on description object `IntPairToDouble`). However, both description objects have the same high-level type, the integer pair. Therefore, the high-level type is used as a middle ground to transform between the two representation types:

```scala
var x: @repr(IntPairToLong) (Int, Int) = 
  toRepr(IntPairToLong, (3, 5))
val y: @repr(IntPairToDouble) (Int, Int) = 
  toRepr(IntPairToDouble, (2, 6))

x = toRepr(IntPairToLong, toHigh(IntPairToDouble, y))
```

Finally, the COMMIT phase transforms the example to:

```scala
var x: Long = IntPairToLong.toRepr(3, 5))
val y: Double = IntPairToDouble.toRepr(2, 6))

x = IntPairToLong.toRepr[IntPairToDouble.toHigh(y)]
```
Therefore, the value \( x \) is converted from a double to a pair of integers, which is subsequently converted to a long integer. This shows the disciplined way in which different \texttt{adrt} scopes compose, allowing values to flow across different representations, from one scope to another. Let us now look at the second scenario.

**Different transformation scopes can be safely nested** and the high-level types are correctly isolated:

```scala
(\texttt{Int, Int}) = (0, 1)
\texttt{val}
\texttt{def}
\texttt{IntPairAsLong}
\texttt{println}(x.toString)
\texttt{adrt}(\texttt{IntPairComplexToLongComplex})
\texttt{var}
\texttt{^}
\texttt{val}
\texttt{n1:}
\texttt{\texttt{//} y = x}
\texttt{var}
\texttt{y1: (Int, Int) = (0, 1)}
\texttt{\texttt{//} y = x}
\texttt{\texttt{//} y = 123.toLong}
```

Values of the high-level types in the inner scope are independently annotated and are transformed accordingly. Since both the integer and the float pairs are encoded as long integers, a natural question to ask is whether values can leak between the two high-level types, for example, by uncommenting the last two lines of the inner scope. This would open the door to incorrectly interpreting an encoded value as a different high-level type, introducing unsoundness.

The answer is no: the code is first type-checked against the high-level types even before the \texttt{INJECT} transformation has a chance to annotate it. This prohibits direct transfers between the high-level types and their representations. Thus, the unsound assignments will be rejected, announcing the programmer that the types do not match. This is a non-obvious benefit of using the ADR transformation instead of manually refactoring the code and using implicit conversions, which would allow such unsound assignments.

**Handling conflicting nested transformation description objects** is another important property of composition:

```scala
adrt (PairAsMyPair) {
  adrt (IntPairAsLong) {
    adrt (FloatPairAsLong) {
      \texttt{val} x: (Float, Float) = (1f, 0f)
      \texttt{var} y: (Int, Int) = (0, 1)
      \texttt{\texttt{//} y = x}
      \texttt{\texttt{//} y = 123.toLong}
    }
  }
  println(x.toString)
}
```

In the code above, the type of \( x \) matches both transformation description objects, so it could be transformed to both representation types \texttt{MyPair[Int, Int]} and \texttt{Long}. However, during the \texttt{INJECT} phase, if a value is matched by several nested transformation description objects, this can be reported to the programmer either as an error or, depending on the implementation, as a warning, followed by choosing one of the transformation description objects for the value:

```scala
adrt (PairAsMyPair) {
  adrt (IntPairAsLong) {
    adrt (FloatPairAsLong) {
      \texttt{val} x: (Int, Int) = (2, 3)
    }
  }
  println(x.toString)
}
```

Furthermore, since the \texttt{INJECT} phase annotates value \( x \) with the chosen transformation, there will be no confusion in the next line, where \( x \) has to be converted back to the high-level type to receive the \texttt{toString} method call, despite the fact that the \texttt{adrt} scope surrounding the instruction uses a different transformation description object.

**Prohibiting access to the representation type inside the transformation scope is limiting.** For example, a performance-conscious programmer might want to transform the high-level integer pair into a floating-point pair without allocating heap objects. Since the programmer does not have direct access to the representation, it looks like the only solution is to decode the integer pair into a heap object, convert it to a floating-point pair and encode it back to the long integer.

There is a better solution. As we will later see, the programmer can use bypass methods to “serialize” the integer pair into a long integer and “de-serialize” it into a floating-point pair. Yet, this requires a principled change in the transformation description object. This is the price to pay for a safe and automated representation transformation.

The main insight of this section is that focusing on individual values and storing the transformation semantics in the annotated type allows us to correctly handle values flowing across scopes, a property we call scope composition. Although we focused on values, method parameters and return types can be annotated in exactly the same way. The next part extends scope composition across separate compilation.

### 4.3 Separate Compilation

Annotating the high-level type by the transformation semantics allows different \texttt{adrt} scopes to seamlessly pass encoded values. To reason about composing scopes across different compilation runs, let us assume we already compiled the \texttt{gcd} method in the motivating example:

```scala
\texttt{adrt (PairAsMyPair)} {
  \texttt{def gcd} (n1: (Int,Int), n2: (Int,Int)): (Int,Int) = ..
}
```

After the \texttt{INJECT} phase, the signature for method \texttt{gcd} is:

```scala
\texttt{adrt (PairAsMyPair)} {
  \texttt{def gcd} (n1: (Int,Int), n2: (Int,Int)): (Int,Int) = ...
}
```

And, after the entire compilation pipeline transformed the code, the bytecode signature for method \texttt{gcd} becomes:

```scala
\texttt{adrt (PairAsMyPair)} {
  \texttt{def gcd}(n1: Long, n2: Long): Long = ...
}
```

When compiling source code that refers to existing low-level code, such as object code or bytecode compiled in a previous run, compilers need to load the high-level signature of each symbol. For C and C++ this is done by parsing header files while for Java and Scala, it is done by reading the source-level signature from the bytecode metadata. However, not being aware of the ADR transformation of method \texttt{gcd}, a separate compilation could assume it accepts two pairs of integers as input. Yet, in the bytecode, the \texttt{gcd} method accepts \texttt{Long}s and is not able to handle pairs of integers.

The simplest solution is to create two versions for each transformed method: the transformed method itself and a bridge, which corresponds to the high-level signature. The bridge method would accept pairs of integers and encode...
them as longs before calling the transformed version of the \texttt{gcd} method. It would also decode the result of \texttt{gcd} back to a pair of integers. This approach allows calling \texttt{gcd} from separately compiled files without being aware of the transformation. Still, we can do better.

**Persisting transformation annotations.** Let us assume we want to call the \texttt{gcd} method from a scope transformed using the same transformation description object as we used when compiling \texttt{gcd}, but in a different compilation run:

```scala
adr\texttt{[IntPairComplexToLongComplex]} { 
  val n1: (Int, Int) = ...
  val n2: (Int, Int) = ...
  val res: (Int, Int) = \texttt{gcd}(n1, n2)
}
```

In this case, would it make sense to call the bridge method? The values \texttt{n1} and \texttt{n2} are already encoded, so they would have to be decoded before calling the bridge method, which would then encode them back. This is suboptimal. Instead, what we want is to let the \texttt{adr\texttt{t}} scopes become part of the high-level signature, but without making the transformation a first-class language feature. To do this, instead of persisting the scope, we persist the injected annotations, including the reference to the transformation description object. They become part of the signature of \texttt{gcd}:

```scala
// loaded signature for method \texttt{gcd}:
// def gcd(n1: \texttt{repr(.) (Int, Int)},n2: \texttt{repr(.) (Int, Int)}):\texttt{repr(.) (Int, Int)}
```

The annotations are loaded just before the \texttt{INJECT} phase, which transforms our code to:

```scala
val n1: \texttt{repr(.) (Int, Int)} = ...
val n2: \texttt{repr(.) (Int, Int)} = ...
val res: \texttt{repr(.) (Int, Int)} = \texttt{gcd}(n1, n2)
```

With the complete signature for \texttt{gcd}, the \texttt{COERCER} phase does not introduce any coercions, since the arguments of method \texttt{gcd} are going to be transformed the same way as the method parameters have been transformed in a previous compilation run. This allows \texttt{adr\texttt{t}} scopes to seamlessly compose even across separate compilation. After the \texttt{COMMIT} phase, the scope is compiled to:

```scala
val n1: \texttt{Long} = ...
val n2: \texttt{Long} = ...
val res: \texttt{Long} = \texttt{gcd}(n1, n2) // no coercions!!!
```

**Making bridge methods redundant.** Persisting transformation information in the high-level signatures allows us to skip creating bridges. For example, calling the \texttt{gcd} method outside the \texttt{adr\texttt{t}} scope is still possible:

```scala
val res: (Int, Int) = \texttt{gcd}((55, 2), (17, 13))
```

Since the signature for method \texttt{gcd} references the transformation description object, the \texttt{COERCER} phase knows exactly which coercions are necessary:

```scala
val res: (Int, Int) = \texttt{toHigh}(....,
  \texttt{gcd}(\texttt{toRepr}(..., (55, 2), \texttt{toRepr}(..., (17, 13))))
```

The main insight in this section is that persisting the description object within each value’s annotated type allows efficient scope composition across separate compilation runs.

### 4.4 Optimizing Method Invocations

When choosing a generic container, such as a pair or a list, programmers are usually motivated by the very flexible interface, which allows them to quickly achieve their goal by invoking the container’s many convenience methods. The presentation so far focused on optimizing the data representation, but to obtain peak performance, the method invocations need to be transformed as well:

```scala
adr\texttt{[IntPairComplexToLongComplex]} {
  val n = (0, 1)
  println(n.toString)
}
```

When handling method calls where the receiver encoded, the default LDL behavior is very conservative: it decodes the value back to its high-level type, which exposes the original method and generates a dynamically-dispatched call (§3.2):

```scala
val n: \texttt{Long} = ...
println([IntPairComplexToLongComplex].\texttt{bypass\texttt{toString}(n)})
```

The bypass method can operate directly on the encoded version of the integer pair, avoiding a heap allocation. In practice, when the receiver of a method call is annotated, our modified LDL transformation looks up the \texttt{bypass\texttt{toString}} method in the transformation description object, and, if nothing is found, warns the programmer and proceeds with generating the dynamically-dispatched call.

**Methods added via implicit conversions, or other enrichment techniques, such as extension methods or type classes add another layer or complexity.** Let us look at the the multiplication operator \(*\) in the first example in the paper, which is added via an implicit conversion (we will further analyze the interaction with implicit conversions in §4.5):

```scala
adr\texttt{[IntPairComplexToLongComplex]} {
  val n1 = (0, 1)
  val n2 = n1 * n1
  ...
}
```

Type-checking the program introduces an explicit call to the implicit conversion that adds the \(*\) operator:

```scala
val n1: (Int, Int) = (0, 1)
val n2: (Int, Int) = \texttt{IntPairIsComplex(n1)} * n1
```

This is a costly pattern, requiring \texttt{n1} to be decoded into a pair and passed to the \texttt{IntPairIsComplex} method, which itself creates a wrapper object that exposes the \(*\) operator. To optimize this pattern, the ADR transformation looks for a bypass method in the transformation description object that corresponds to a mangled name that combines the implicit method and the operator. For simplicity, let us assume its name is \texttt{implicit\_+}:

```scala
val n1: \texttt{Long} = IntPairComplexToLongComplex.toRepr(0, 1)
val n2: \texttt{Long} = implicit\_+(n1, n1)
```
This allows the call to the + operator to go through without any heap object creation, which significantly improves the performance and the heap footprint. Yet, we glossed over some details in how bypass methods are implemented.

**Bypass methods.** Both normal and implicit bypass methods need to correspond to the method they are replacing and:

- Add a first parameter corresponding to the receiver;
- Have the rest of the parameters match the method;
- Freely choose parameters to be encoded or decoded.

Therefore, during the COERC phase, which introduces extension methods, the implicit_ method has the signature:

```scala
def implicit_*(recv: Int, n2: Int, @repr (Int, Int)): @repr (Int, Int)
```

Since the method is free to choose the encoding of its arguments, the following signature would also be accepted, but would be suboptimal:

```scala
def implicit_*(recv: (Int, Int), n2: (Int, Int)): (Int, Int)
```

It is interesting to notice that representation-specific method rewriting relies on two previous design choices: (1) shifting focus from scopes to individual values and (2) carrying the entire transformation semantic in the signature of each encoded value.

### 4.5 Interaction with Other Language Features

This section presents the interaction between the ADR transformation and object-oriented inheritance, generics and implicit conversions, explaining what are the additional steps that must be taken to ensure correct program transformation and what are the limitations of our approach.

**Dynamic Dispatch and Overriding** is an integral part of the object-oriented programming model, allowing objects to encapsulate code. The main approach to evolving this encapsulated code is extending the class and overriding its methods. However, changing the data representation can lead to situations where source-level overriding methods are no longer overriding in the low-level bytecode:

```scala
class X{
  def identity(i: (Int, Int)): (Int, Int) = i
  @bridge (IntPairAsLong) {
    override def identity(i: (Int, Int)) = t
  }
}
```

In the low-level bytecode, the identity method in class \( Y \) no longer overrides method identity in class \( X \), as its low-level signature expects a long integer instead of a pair of integers. This prompted us to extend the Late Data Layout mechanism, introducing a new BRIDGE phase, which runs just before COERC and introduces bridge methods to enable correct overriding. After the INJECT phase, the code corresponding to class \( Y \) is:

```scala
class Y extends X(t: @repr(...))(Int, Int) {
  def identity(i: @repr(...))(Int, Int): @repr(...)(Int, Int) = t
  @bridge override def identity(i: (Int, Int)) = t
}
```

The BRIDGE phase inserts the methods necessary to allow correct overriding (return types are omitted):

```scala
class Y extends X(t: @repr(...))(Int, Int) {
  def identity(i: @repr(...))(Int, Int) = t
  @bridge // overrides method identity from class X:
  override def identity(i: (Int, Int)) = identity(i)
}
```

The COERC and COMMIT phases then transform class \( Y \) as before, resulting in a class with two methods, one containing the optimized code and another that overrides the method from class \( X \), marked as @bridge:

```scala
class Y extends X(t: Long) {
  def identity(i: Long): Long = t
  @bridge override def identity(i: (Int, Int)) = IntPairAsLong-toHigh(identity(toRepr(i))
}
```

If we now try to extend class \( Y \) in another adrt scope with the same transformation description object, overriding will take place correctly; the new class will define both the transformed method and the bridge, overriding both methods above. However, a more interesting case is extending class \( Y \) from a scope with a different description object:

```scala
adrt(IntPairAsDouble) {
  class Z(t: (Int, Int)) extends Y(t) {
    override def identity(i: (Int, Int)) = i
  }
}
```

The ensuing BRIDGE phase generates 2 bridge methods:

```scala
class Z(t: Double) extends Y(...) {
  def identity(i: Double) = i
  @bridge override def identity(i: (Int, Int)) = ... 
  @bridge override def identity(i: Long) = ...
}
```

Although the resulting object layout is consistent, the @bridge methods use coercions to transform between the representations, which makes them less efficient. This is even more problematic when up-casting class \( Z \) to \( Y \) and invoking identity, as the bridge method goes through the high-level type to convert the long integer to a double. To make the programmer aware, the BRIDGE phase issues warnings if more than one representation needs to be bridged.

**Generics** Another problem that arises when performing ad hoc programmer-driven transformations is how to transform the data representation in generic containers. Should the ADR transformation be allowed to change the data representation stored in a List? We can use an example:

```scala
def use1(list: List[(Int, Int)]): Unit = ...
adrt(IntPairAsLong) {
  def use2(list: List[(Int, Int)]): Unit = use1(list)
}
```

In the specific case of the Scala immutable list, it would be possible to convert the list parameter of use2 from type List[Long] to List[(Int, Int)] before calling use1. This can be done by mapping over the list and transforming the representation of each element. However, this domain-specific knowledge of how to transform the collection only applies to the immutable list in the standard library, and not to other generic classes that may occur in practice.
Furthermore, there is an entire class of containers for which this approach is incorrect: mutable containers. An invariant of mutable containers is that any elements changed will be visible to all the code that holds a reference to the container. But duplicating the container itself and its elements (stored with a different representation) breaks this invariant: changes to one copy of the mutable container are not visible to its other copies. This is similar to the mutability restriction in §4.1.

The approach we follow in the ADR transformation is to preserve the high-level type inside generics. Thus, our example after the COMMIT phase will be:

```scala
def use1(list: List[(Int, Int)]): Unit = use1(list)
```

Still, this does not prevent a programmer from defining another transformation description object that targets List[Int] and replaces it by List[Long]:

```scala
val a: Any = n
println(a)
```

After the COMMIT phase, the transformation produces:

```scala
val a: Any = n
println(a)
```

Therefore `adr` scopes are capable of targeting:
- generic types, such as `List[T]` for any `T`;
- instantiated generic types, such as `List[(Int, Int)];`
- monomorphic types, such as `(Int, Int)`.

Using these three cases and scope composition, programmers can conveniently target any type in their program.

Implicit Conversions interact in two ways with `adr` scopes:

*Extending the object functionality* through implicit conversions or any other means, such as extension methods or type classes must be taken into account by the bypass method call rewriting in the COERCES phase. The handling of all three means of adding object functionality is similar, since in all three cases, the call to the new method needs to be intercepted and redirected. Depending on the exact means, the mangled name for the bypass method will be different, but the mechanism and signatures remain the same (§4.4).

*Offering an alternative to the LDL-based backend.* Unfortunately, implicit conversions do not offer strong enough guarantees to replace the LDL backend. For example, assuming there are implicit methods in scope to transform pairs of integers to longs and back, we can try to transform:

```scala
val n: (Int, Int) = (1, 0)
printLN(n)
```

To trigger the transformation, we replace the type of `n`:

```scala
val n: Long = implicitIntPairToLong((1, 0))
```

This resulting code changes semantic because no coercion is applied to `a`, since `Long` is a subtype of `Any`. In turn, this makes the output `4294967296` instead of `(1, 0)`. As we have seen in §3, the missing coercions is correctly inserted when annotations track the value representation, since annotations are orthogonal to the host language type system.

With this, we presented the main insights in the Ad hoc Data Representation Transformation approach and how they interact with other language features to guarantee transformation correctness. The next section describes the architecture and implementation of our Scala compiler plugin.

5. Implementation

This section describes the technical aspects of our implementation that can aid a compiler developer in porting the approach to another language. We implemented the ADR transformation as a Scala compiler plugin [2], by extending the open-source multi-stage programming transformation provided with the LDL [41] artifact, available at [4].

The `adr` scope acts as the trigger for the ADR transformation. We treat it as a special keyword that we transform immediately after parsing, in the POSTPARSER phase. To show this, we follow a program through the compilation stages:

```scala
def using1(list: List[(Int, Int)]): Unit = ...
```

Immediately after the source is parsed, the POSTPARSER phase transforms the `adr` scopes in three steps:
- it attaches a unique id to each `adr` scope;
- it records and clears the block enclosed by the `adr` scope;
- it inlines the recorded code immediately after the non-empty `adr` scope and, in the process, it marks the value and method definitions by the `adr` scope’s unique id (or by multiple ids, if `adr` scopes are nested).

Following the POSTPARSER phase, the code is:

```scala
def using1(list: List[(Int, Int)]): Unit = ...
```

This code is ready for type-checking: the definition of `n` is located in the same block as its use, making the scope correct. During the type-checking process, the `IntPairToLong` object is resolved to a symbol, missing type annotations are inferred and implicit conversions are introduced explicitly in the tree. After type-checking and pattern matching expansion, the INJECT phase traverses the tree and:
- for every `adr` scope it records the id and description object, before removing it from the abstract syntax tree;
- for value and method definitions, if the type matches one or more transformations, it adds the `@repr` annotation.

Following the INJECT phase, the code for our example is:

```scala
def using1(list: List[(Int, Int)]): Unit = ...
```

After the INJECT phase, the annotated signatures are persisted, allowing the scope composition to work across sep-
rate compilation. Later, the `BRIDGE`, `COERC`e and `COMM`IT phases proceed as described in §3 and §4.

The **transformation description objects** extend the marker trait `TransformationDescription`. Although the marker trait is empty, the description object needs to define at least the `toList` and `toRepr` coercions, which may be generic (§4.1). The programmer is then free to add bypass methods, in order to avoid decoding the representation type for the purpose of dynamically dispatching method calls. To aid the programmer in adding bypass methods, the `COERC`e phase warns whenever it does not find a suitable bypass method, indicating both the expected name and the expected method signature. Here we encountered a bootstrapping problem: although bypass methods handle the representation type, during the `COERC`e phase, their signatures are expected to accept parameters of the annotated high-level type, in order to allow redirecting method calls. To work around this problem, we added a the `@high` annotation, which acts as an anti-`@repr` and marks the representation types:

```scala
object IntPairToLong extends TransformationDescription{
  def bypass_toString(repr: @high Long): String = ...
}
```

This mechanism allows programmers to both define and use the transformation description objects in the same compilation run. Considering the difficult nature of bootstrapping transformations, we are content with the current solution. Another advantage we get for free, thanks to referencing the transformation description object in the type annotation, is an explicit dependency between all transformed values and their description objects. This allows the Scala incremental compiler to automatically recompile all scopes when the description object in their `adrt` marker has changed.

This concludes the section, which explained how we solved the two main technical problems in the ADR Transformation and how this impacted the compilation pipeline.

6. **Benchmarks**

This section evaluates the experimental benefits of ADR transformations and shows four different usage scenarios.

6.1 **Setup**

We ran the benchmarks on an Intel 17-4702HQ quad-core processor machine with the frequency fixed at 2.2GHz. The RAM available to the benchmarks was 2GB and the running times were measured using the scalamer benchmarking platform [30], to avoid noise.

6.2 **ADRT Scope Benchmarks**

We chose the benchmarks in order to cover a wide range of transformations using the `adrt` scope:

- the greatest common divisor algorithm, presented in §2;
- least squares benchmark + deforestation [42];
- averaging sensor readings + array of struct;
- computing the first 2000 Hamming numbers.

The benchmarks were optimized using our implementation of the `adrt` scope at [2] and are described in detail on the website [3]. We will proceed to explain the transformation in each benchmark, but, due to space constraints, the full descriptions are only available on the website.

The **Gaussian Greatest Common Divisor** is the running example described in §2 and used throughout the paper. It is a numeric and CPU-bound benchmark, where the main slowdown is caused by heap allocations.

The `adrt` transformation surrounds the tail-recursive `gcd` method and optimizes the pair of integer representation of Gaussian integers by encoding them in long integers. From this point of view, we can think of the transformation as retrofitting the value class status to the pair of integers. The benchmark results in Table 1 show a 13x speed improvement and we checked that no object allocations occur in the transformed version of the `gcd` method. The transformation description object is 30 lines of code.

The **Least Squares Method** takes a list of points in two dimensions and computes the slope and offset of a straight line that best approximates the input data. The benchmark performs multiple traversals over the input data and thus benefits from deforestation [42], which avoids the creation of intermediate collections after each `map` operation:

```scala
object LeastSquares extends TransformationDescription {
  def leastSquares(data: List[(Double, Double)]) = {
    val size = data.length
    val sumx = data.map(_._1).sum
    val sumy = data.map(_._2).sum
    val sumxy = data.map(p => p._1 * p._2).sum
    val sumxx = data.map(p => p._1 * p._1).sum
    ...
  }
}
```

The `adrt` scope performs a generic transformation from `List[T]` to `LazyList[T]`:

```scala
object ListAsLazyList extends TransformationDescription {
  def toRepr[T](list: List[T]): LazyList[T] = ...
  def toHigh[T](list: LazyList[T]): List[T] = ...
  // Bypass methods
}
```

The `LazyList` collection achieves deforestation by recording the mapped functions and executing them lazily, either when `force` is invoked on it or when a `fold` operation is executed. Since the `sum` operation is implemented as a `foldLeft`, the `LazyList` applies the function and sums the result without creating an intermediate collection. This transformation alone produced a 3x speedup for an input of 5 million points and made the `adrt` transformation perform better than the dedicated collection optimization tool `scalablitz` [7, 12], which produced a modest 1.7x speedup.

Yet, using the `LazyList` collection we can also benefit from specialization [19]. Using a specialized version of the `LazyList` collection we obtained a 5x speed improvement thanks to a combination of deforestation and specialization. Therefore, we used the `adrt` transformation both to trans-
form the collection semantics and to retrofit specialization in a localized scope. The transformation description object has 30 lines of code and the LazyList has 70 lines of code.

The Sensor Readings benchmark was inspired by the Sparkle visualization tool [9] that is able to quickly display, zoom, transform and filter sensor readings. To obtain nearly real-time results, Sparkle combines several optimizations such as streaming and array of struct to struct of array conversions, all currently implemented by hand. In our benchmark, we implemented a mock-up of the processing core of Sparkle and automated the array of struct optimization:

```scala
val adrt(BigIntToLong) {
  def toRepr(soa: StructOfArray): SensorReadings = ...
  def toHigh(soa: StructOfArray): SensorReadings = ...
}
```

Using the adrt scope produced a modest speedup of 2.2x, which initially puzzled us, since we were expecting better speedups. When we implemented the transformation by hand, manually refactoring the code, we obtained a speedup of 2.25x, showing this is the best that can be achieved for the transformation. Still, this shows the adrt scope is able to change the underlying data structures and provide speedups while maintaining program semantics. The transformation description object has 60 lines of code.

The Hamming Numbers Benchmark computes numbers that only have 2, 3 and 5 as their prime factors, in order. Unlike the other benchmarks, this is an example we randomly picked from Rosetta Code [6] and attempted to speed up:

```scala
import scala.collection.mutable.Queue
adrt[BigIntToLong] {
  class StructOfArray[arrayOfTimestamps: Array[Long],
   arrayOfEvents: Array[Long],
   arrayOfReadings: Array[Double]]
  object AoSToSoA extends TransformationDescription {
    def toRepr(aos: SensorReadings): StructOfArray = ...
    def toHigh(soa: StructOfArray): SensorReadings = ...
  }
}
```

An observation is that, for the first 10000 Hamming numbers, there is no need to use BigInt, since the numbers fit into a Long integer. Therefore, we used two nested adrt scopes to replace BigInt by Long and Queue[BigInt] by a fixed-size circular buffer built on an array. The result was an 8x speedup. The main point in the transformation is its optimistic nature, which makes the assumption that, for the Hamming numbers we plan to extract, the long integer and a fixed-size circular buffer are enough. This is an example of using local domain-specific knowledge to optimize the program. Should the optimistic assumption fail, the transformed code will produce exceptions that announce the optimistic assumptions are invalidated. For this example, the size of the transformation description objects is 100 lines of code.

### 6.3 Scoped Library Replacement

Before the adrt scope was developed, we worked on a plugin that used a similar scoped approach: an extension used by the miniboxing transformation [5, 40] to replace library functions and tuples by custom optimized versions. Although this plugin was a precursor of the adrt transformation, being scoped but not programmer-driven, we present it because in the meantime it became part of the miniboxing plugin and has been used to transform large pieces of code, proving our approach scales not only to toy examples, but to actual code used in the industry.

The miniboxing transformation [40] proposes an alternative to erasure, allowing generic methods and classes to work efficiently with unboxed primitive types. Unlike the current specialization transformation in the Scala compiler [19], which duplicates and adapts the generic code once for every primitive type, the miniboxing transformation only duplicates the code once and encodes all primitive types in long integers. This allows miniboxing to scale much better than specialization [21] in terms of bytecode size while providing comparable performance. Yet, one of the main drawbacks of using the miniboxing plugin is that all Scala library classes are either generic or specialized with the built-in Scala specialization scheme, which is not compatible with miniboxing. Therefore, interacting with functions and tuples from minboxed code incurs significant overhead.

The Scala programming language offers functions as first-class citizens. However, since functions are not first-class citizens in the Java Virtual Machine bytecode, the Scala compiler desugars them to anonymous classes extending a functional interface. The following example shows the desugaring of function $(x: Int) \to x + 1$:

```scala
class $anon extends Function1[Int, Int] {
  def apply(x: Int): Int = x + 1
}
```

### Table 1. Benchmark running time for each use case.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original</th>
<th>ADRT</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian GCD</td>
<td>3.05 µs</td>
<td>0.23 µs</td>
<td>13x</td>
</tr>
<tr>
<td>Least Sq. Blitz (5M)</td>
<td>8026 ms</td>
<td>4763 ms</td>
<td>1.7x</td>
</tr>
<tr>
<td>Least Sq. adrt 1 (5M)</td>
<td>8026 ms</td>
<td>2393 ms</td>
<td>3x</td>
</tr>
<tr>
<td>Least Sq. adrt 2 (5M)</td>
<td>8026 ms</td>
<td>1643 ms</td>
<td>5x</td>
</tr>
<tr>
<td>Sensor Readings (5M)</td>
<td>50.8 ms</td>
<td>23.1 ms</td>
<td>2x</td>
</tr>
<tr>
<td>Hamming 10000th</td>
<td>4.35 ms</td>
<td>0.55 ms</td>
<td>8x</td>
</tr>
</tbody>
</table>
This function desugaring does not expose a version of the apply method that encodes the primitive type as a long integer, as the miniboxing transformation expects. Therefore, when programmers write minboxed code that uses functions, they have two choices: either accept the slowdown caused by converting the representation or define their own MiniboxedFunction1 class, and perform the function desugaring by hand. Neither of these is a good solution.

What we would like to have is a way to transform the references to Function1 in minboxed code: instead of extending Function1, anonymous functions should extend MiniboxedFunction1. But the problem is that the minboxed code needs to interoperate with library-defined code, or with other libraries that were not transformed. This is where the scope comes in: the minboxed code acts as a scope for the function and tuple representation transformation, i.e., the ADR transformation of Function and Tuple. We will now focus on two published benchmarks that exercise the function and tuple optimization.

The Scala-Streams library draws its inspiration from Java 8 streams and other libraries. It has been described in the literature [13] and is available as an open-source implementation [1]. In its continuation-based design, each stream combinator provides a function that is stacked to form a transformation pipeline. As the consumer reads from the final stream, the transformation pipeline is executed, processing an element from the source into an output element. However, the pipeline architecture is complex, since combinators such as filter may drop elements, stalling the pipeline. This makes the Scala Streams an interesting platform to study the performance benefits of the miniboxing transformation and, in turn, of our adrt precursor.

Without going into the details of the benchmarks, which are covered in [13], Table 2 shows the results with and without our adrt precursor extension, showing up to 14.5x speedups when functions are optimized.

The Framian Vector implementation is an exploration into deeply specializing the immutable Vector bulk storage without using reified types [10, 11]. This is a benchmark performed by a commercial specialization/miniboxing user in the Scala community. Table 3 shows a 4.4x speed improvement when the function representation is optimized.

We will close the section by concluding that the adrt scopes are capable of covering a broad range of custom and scoped transformations and the technique has been shown to scale to large programs, through the miniboxing plugin extension based on the adrt approach.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Generic</th>
<th>Minboxed</th>
<th>Minboxed +functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum</td>
<td>100.6 ms</td>
<td>355.9 ms</td>
<td>12.0 ms</td>
</tr>
<tr>
<td>SumOfSquares</td>
<td>188.3 ms</td>
<td>450.9 ms</td>
<td>13.0 ms</td>
</tr>
<tr>
<td>SumOfSqEven</td>
<td>130.8 ms</td>
<td>300.4 ms</td>
<td>52.2 ms</td>
</tr>
<tr>
<td>Cart</td>
<td>220.6 ms</td>
<td>560.2 ms</td>
<td>55.3 ms</td>
</tr>
</tbody>
</table>

Table 2. Scala Streams pipelines for 10M elements.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual C-like code</td>
<td>0.650 µs</td>
</tr>
<tr>
<td>Miniboxing with functions</td>
<td>0.705 µs</td>
</tr>
<tr>
<td>Miniboxing without functions</td>
<td>3.080 µs</td>
</tr>
<tr>
<td>Generic</td>
<td>13.409 µs</td>
</tr>
</tbody>
</table>

Table 3. Mapping a 1K vector.

7. Related Work

Changing data representations is a well-established and time-honored programming need. Techniques for removing abstraction barriers appear in the literature since the invention of high-level programming languages and often target low-level data representations. However, our technique is distinguished by its automatic determination of when data representations should be transformed, while giving the programmer control of how to perform this transformation and on which scope it is applicable. We survey some recent related work.

As discussed earlier, the standard optimizations that are closest to our approach are value classes [8] and class specialization [19, 40]. These are optimizations with great practical value, and most modern languages have felt a need for them. For instance, specialization optimizations have recently been proposed for adoption in Java, with full VM support [22]. Rose has an analogous proposal for value classes [32, 33] in Java. Unlike our approach, all the above are whole-program data representation transformations and receive limited programmer input (e.g., a class annotation).

Virtual machine optimizations often also manage to produce efficient low-level representations through tracing [20] or inlining and escape analysis [18, 35]. Furthermore, modern VMs, such as V8, Truffle [43] and PyPy [14] attempt specialization and inference of optimized layouts. However, the ability to perform complex inferences dynamically is limited, and there is no way to draw domain-specific knowledge from the programmer. Generally VM optimizations are often successful at approaching the efficiency of a static language in a dynamic setting, but not successful in reliably exceeding it.

In terms of transformations, we already presented the Late Data Layout [41] mechanism in the Scala setting. Similar approaches, with different specifics in the extent of type system and customization support, have been applied to Haskell [24]. Foundational work exists for ML, with Leroy [27] presenting a transformation for unboxing objects, with the help of the type system. Later work extends [38] and generalizes [34] such transformations. In terms of transformations runtime-dispatched generics, we refer to the work on Napier88 [29] and the TIL compiler [37] [23].

In the specific setting of data structure specialization, the CoCo approach [44] adaptively replaces uses of Java collections with optimized representations. CoCo has a similar high-level goal as our techniques, yet focuses explicitly on collections only. Approaches that only target a finite number of classes (data structure implementations) can be realized entirely in a library. An adaptive storage strategy for Python
collections [15], for instance, switches representations once collections become polymorphic or once they acquire many elements.

Multi-stage programming [36] is another technique that optimizes the data representation. Its Scala implementation, dubbed lightweight modular staging and can both optimize and even re-target parts of a program to GPUs [16, 31]. Yet, multi-stage programming scopes are not accessible from outside, making it impossible to call a transformed method or read a transformed value. Instead, the transformation scope is closed and nothing is assumed to be part of the interface. Hopefully, this will be improved by techniques such as the Yin-Yang staging front-end [25], based on Scala macros [17]. Another type-directed transformation in the Scala compiler is the picking framework [28], also based on macros. Instead of transforming the data representation in-place, pickler combinators create serialization code that allows can efficiently convert an object to a wide range of formats.

8. Conclusion

In this paper we presented an intuitive interface over a safe and composable programmer-driven data representation transformation, where the composition works not only across source files but also across separate compilation runs. The transformation takes care of all the tedium involved in using a different representation, by automatically introducing coercions and bridge methods where necessary and optimizing the code via extension methods. Benchmarking the resulting transformation shows significant performance improvements, with speedups between 1.9x and 13x.

References