Automating Ad hoc Data Representation Transformations

Vlad Ureche[†] Aggelos Biboudis[*] Yannis Smaragdakis[*] Martin Odersky[†]

[†]École polytechnique fédérale de Lausanne, Switzerland: `{first.last}@epfl.ch`
[*]University of Athens, Greece: `{biboudis, smaragd}@di.uoa.gr`

Abstract

To maximize run-time performance, programmers often specialize their code by hand, replacing library collections and containers by custom objects in which data is restructured for efficient access. However, changing the data representation 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 mechanism that allows programmers to safely change the data representation in delimited scopes containing anything from expressions to entire class definitions. To achieve this, programmers define a transformation and our mechanism automatically and transparently applies it during compilation, eliminating the need to manually change the source code.

Our technique leverages the type system in order to offer correctness guarantees on the transformation and its interaction with 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 all the way to collection deforestation. On our benchmarks, the technique obtained speedups between 1.8x and 24.5x.

Categories and Subject Descriptors E.2 [*Object representation*]

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:

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

Unlike evolving the interface, there is no mechanism in modern languages for evolving an object's encapsulated data as the programmer sees fit. The encapsulated data format 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 our 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, avoiding the memory allocation and thus the garbage collection cost. Additionally, what if we could also add functionality, such as arithmetic operations, directly on our ad hoc complex numbers, without any heap allocation overhead?

Object layout transformations are common in dynamic language virtual machines, such as V8, PyPy and Truffle. These virtual machines profile values at run-time and make optimistic assumptions about the shape of objects. This allows them to improve the object layout in the heap, at the cost of recompiling all of 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 significant overheads. Thus, runtime decisions to change the low-level layout are 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

OOPSLA '15, October 25-30 2015, Pittsburgh, PA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3689-5/15/10...$15.00.
http://dx.doi.org/10.1145/2814270.2814271

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 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 for each pair. Even worse, specialization and value class inlining are mutually exclusive, making it impossible to optimally represent our "complex numbers" even if we had complete control over the Scala library. Furthermore, our encoded "complex number" data representation may be applicable for specific parts of the client code, but might not make sense globally.

In our "complex numbers" abstraction, we only use a fraction 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 encoded representation, effectively specializing that part of our program.

In this paper we present such an automated transformation that allows programmers to safely change the data representation in limited, well-defined scopes that can include anything from expressions to method and class definitions. The transformation, which occurs during compilation, maintains strong correctness guarantees in terms of non-orthogonal language features, such as dynamic dispatch, inheritance and generics, while also maintaining consistence across separate compilation runs.

Like metaprogramming, which allows developers to transform their code in an ad-hoc ways, our technique allows redefining the data representation to be used inside delimited scopes. Because of its power, the technique also affords potential for misuse. In some cases, specifically for mutable and reference-based data structures, the transformations must be carefully designed to preserve language semantics (§4.5). Still, altering program semantics may be desirable—we exploit this property in the deforestation benchmark, shown in the evaluation section (§6).

The scoped nature of the transformation tightly controls which parts of the code use the new data representation and operations while the mechanism for defining transformations automatically eliminates many of the common semantics-altering pitfalls. Given a programmer-designed data representation transformation, inside the delimited scopes the compiler is responsible for: (1) automatically deciding when to apply the transformation and when to revert it, in order to ensure correct interchange between representations, (2) enriching the transformation with automatically generated bridge code that ensures correctness relative to overriding and dynamic 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 *composability* in the following sections. Thus, our approach adheres to the design principle of separating the reusable, general and provably correct transformation *mechanism* from the programmer-defined *policy*, which may contain incorrect decisions [31]. In this context, our main contributions are:

- Introducing the data representation metaprogramming problem, which, to the best of our knowledge, has not been addressed at all in the literature (§2);
- Presenting the extensions that allow global data representation transformations (§3) to be used as scoped programmer-driven transformations (§4);
- Implementing the approach presented as a Scala compiler plugin [3] that allows programmers to express custom transformations (§5) and benchmarking the plugin on a broad spectrum of transformations, ranging from improving the data layout and encoding, to retrofitting specialization and value class status, and to collection deforestation [50]. These transformations produced speedups between 1.8 and 24.5x on user programs (§6).

2. Motivation and Overview

This section presents a motivating example featuring the complex numbers transformation, which we use throughout the paper. It then shows how the data representation transformation is triggered and introduces the main concepts. Finally, it shows a naive transformation, hinting at the difficulties lying ahead.

2.1 Motivating Example

In the introduction, we focused on adding complex number semantics to pairs of integers. Complex numbers with integers as both their real and imaginary parts are known as Gaussian integers [2, 24], 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 result, 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. Furthermore, computing the GCD is similar to Euclid's algorithm for integer numbers:

```
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 on the heap. If we run this algorithm with no optimizations, computing the GCD takes around 3 microseconds (on the same setup as used for our full experiments in §6):

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

Let us now run `gcdADRT`, which has the same code as `gcd` but encodes the Gaussian integers into 64-bit long integers:

```
scala> timed(() => gcdADRT((544, 185), (131, 181)))
The operation takes 0.23 us (based on 10000 executions)
The result is (10, 3).
```

This rather large speedup, of 13x, is the effect of using the long integer representation for Gaussian Integers, which:

(1) Provides a direct representation, which does not require any pointer dereferencing;
(2) Allocates Gaussian integers on the stack, since the `Long` primitive type is unboxed by the compiler backend, thus avoiding object allocation and garbage collector pauses.

The Benchmarks section (§6) shows the contribution of each element to the speedup. This example (and many others in the Benchmarks section) show that optimizing the data representation is worthwhile. However, transforming the code by hand is both tedious and error-prone.

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 is triggered by the `adrt` marker. This marker method accepts two parameters: the first parameter is the *transformation description object* and the second is a block of code that constitutes the *transformation scope*, which can contain anything from expressions all the way to method or even class definitions:

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

The `gcdADRT` method has exactly the same code as `gcd`, but wrapped in the `adrt` scope. Therefore, during compilation, the method is transformed to use the long integer representation. Two elements trigger the transformation: the description object and the transformation scope.

The transformation description object is responsible for defining the transformation that will be applied to the code. In our example, `IntPairComplexToLongComplex` designates a transformation from the *high-level type*, in this case `(Int, Int)` to the *representation type*, in this case `Long`:

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

Transformation description objects are described in more detail in §4, but we can already preview their components:

- The `toRepr` and `toHigh` methods serve a double purpose:
  - At the type level, they define the high-level type, in this case `(Int, Int)`, which serves as the target of the transformation, and the representation type, in this case `Long`, which will be used as the optimized value representation;
  - At the term level, they allow converting values between the two representations;
- The "bypass methods" part of the definition allows operations such as `*`, `%` and `norm` to run directly on values *encoded* in the representation type (in this case `Long`), instead of *decoding* them back to the high-level type in order to execute the dynamic dispatch. We explain how bypass methods are defined and used later on, in §4.4.

Description objects split the task of optimizing the data representation into:

(1) Devising an improved data representation: Defining the improved data representation is done once and uses domain-specific knowledge about the program. Therefore, we let the developer decide how data should be encoded and how operations should be handled. This information is stored in the description object.
(2) Transforming the source code to use the improved representation, based on the description object: This is repetitive, tedious and error-prone work, which we completely automate away.

A natural question to ask is why not automate the process of finding a better data representation? Any change in the data representation speeds up certain patterns at the expense of slowing down others. For example, unboxing primitive types speeds up monomorphic code, which handles primitives directly. Yet, erased generics still require values to be boxed, so any interaction with them triggers boxing operations, which slow down execution.

Furthermore, there are many aspects that can be optimized: eliminating pointer dereferencing, improving cache locality, reducing the memory footprint to avoid garbage collection pauses, reducing numeric value ranges, specializing or delaying operations, and many others. Thus, there are many choices to make, depending on the context, to the point where automation does not make sense. Instead, armed with application profiles and domain-specific information about how the data is used, a programmer can decide what is the best transformation to apply to each critical part of an application. And, interestingly, not all parts of an application have the same needs. This is where scopes come in.

The transformation scope is delimited by the `adrt` marker method, which behaves much like a keyword. Values, methods and classes defined in the scope are also visible outside, since the inlining occurs early in the compilation pipeline:

```
scala> adrt(IntPairComplexToLongComplex) {
     |  def gcdADRT(n1: (Int, Int), n2: (Int, Int))={
     |    ...
     |  }
     | }
defined method gcdADRT

scala> timed(() => gcdADRT((544, 185), (131, 181)))
...
```

Scoped transformations bring two advantages:

- Different parts of a program can use different transformations, using the best data representation for the task;
- Transformations are clearly marked in the source code.

The fact that different transformations can be applied to different components gives the ADR transformation its scoped nature, and sets it apart from classical optimizations such as unboxing primitive types, generic specialization and value class inlining, which occur globally. However, this scoped nature makes the transformation more complex, as the next paragraphs will show.

2.3 A Naive Transformation

Despite its simple interface, the Ad hoc Data Representation Transformation mechanism is by no means simple. Let us try to make the transformation by hand and see the challenges that appear. The initial result, the `gcdNaive` method, would take and return values of type `Long` instead of `(Int, Int)`:

```
def gcdNaive(n1: Long, n2: Long): Long = {
 val remainder = n1 % n2
 if (remainder.norm == 0) n2 else gcdNaive(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 (§4.1)? How and when to encode and decode values, and what to do about values that are visible outside the scope (§4.2)? Even worse, what if parts of the code are compiled separately, in a different compiler run (§4.3)?

Going into the semantics of the program, we can ask if the `%` (modulo) operator maintains the semantics of Gaussian integers when used for long integers. Also, is `norm` defined for long integers? Unfortunately, the response to both questions is negative. Therefore, to correctly transform the code, ADRT needs equivalent versions of the methods that operate on the long integer representation (§4.4).

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 (§4.5):

```
trait WithGCD[T] {
 def gcd(n1: T, n2: T): T
}

class Complex extends WithGCD[(Int, Int)] {
 // expected: gcd(n1: (Int, Int), n2: (Int, Int)) ...
 // found: gcd(n1: Long, n2: Long): Long
 // (which does not implement gcd in trait WithGCD)
 def gcd(n1: Long, n2: Long): Long = ...
}
```

What we can learn from this naive transformation, which is clearly incorrect, is that transforming the data representation is by no means trivial and that special care must be taken when performing it. Our approach, the Ad hoc Data Representation Transformation, addresses the questions above in a reliable and principled fashion.

3. Data Representation Transformations

As necessary background for our approach, we review data representation transformations and, in particular, the Late Data Layout transformation mechanism [47], which we later extend to our Ad hoc Data Representation Transformation.

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 (LDL) 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. However, LDL 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, so later sections will have to extend it.

3.1 Late Data Layout

The Late Data Layout (LDL) mechanism [47] is the underlying transformation used in Scala to implement multi-parameter value class inlining and to specialize classes using the miniboxed encoding [46]. It is a flexible and reliable mechanism, tested on thousands of lines of Scala 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 running example, we have values of types `Int` and `Any`. `Any` is the top of the Scala type system, and thus a supertype of `Int`:

```
val i: Int = 1
val j: Int = i
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:[1]

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

The LDL mechanism transforms the data representation in three phases: INJECT, COERCE and COMMIT. Each of the phases is responsible for a property of the transformation: INJECT makes LDL *selective*, COERCE 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 symbol with its desired representation. In the case of primitive integer unboxing, the annotation is `@unboxed`, and it signals that the value should be stored in the unboxed `int` representation. As an optimization, instead of adding a `@boxed` annotation for the corresponding cases, symbols that are not marked are automatically considered boxed. Following the INJECT phase, the previous example will be transformed to:

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

[1] The translations shown throughout the paper are Scala compiler abstract syntax tree (AST) dumps, in different stages of the compilation. To facilitate reading, we pretty-print the ASTs using Scala syntax. Sometimes we have to introduce elements that are not part of the Scala syntax, such as `int`.

The INJECT phase gives LDL a selective nature, allowing it to mark each individual symbol with its representation. For example, it would have been equally correct if the marking rules decided that `j` should be boxed, in which case it 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 COERCE phase, as its name suggests, introduces coercions. This is done by changing the annotation semantics: annotated types become incompatible with their unannotated 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-typechecking the tree, the COERCE phase can detect representation mismatches and can patch them using coercions. In the example, the last line contains such a mismatch:

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

The COERCE 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:

```
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. The LDL paper [47] 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 [45].

From our perspective, optimality means that once representations are chosen and annotated, the COERCE phase will not introduce any redundant coercions, so data will be seamlessly passed along with as few coercions as possible.

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 unboxed value:

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

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 transformation over the program AST.

3.2 Support For Object-Oriented Programming

The LDL mechanism targets 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 COERCE 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]`. The LDL paper [47] shows 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 extend it to allow the more complex, programmer-driven, Ad hoc Data Representation Transformation.

4. Ad hoc Data Representation Transformation

The Ad hoc Data Representation (ADR) transformation adds two new elements to 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 optimal transformations that may be suboptimal or even incorrect for code outside the 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:[2]

```
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)
  }
}
```

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:

```
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 the corresponding representation type, `Long` and vice-versa and (2) the implementations are called in the transformed scope to encode and decode values as necessary.

Since the description objects can accommodate very different transformations, as shown in the Benchmarks section (§6), we will not attempt to give a recipe for optimizing programs here. Each transformation should be devised by programmers based on runtime profiles and domain-specific knowledge of how data is processed inside the application. Instead, we will focus on the transformation facilities available to the description objects.

Bypass Methods. The description object can optionally include bypass methods, which correspond to the methods exposed by the high-level type, but instead operate on values

[2] In the following paragraphs, the `gcd` method is assumed to be always transformed, so we will skip the `ADRT` suffix, which was used in the Motivation section (§2) to mark the transformed version of the method.

encoded 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 more general, and we describe it 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 ADR transformation is used to target collections regardless of the type of their elements. We analyzed multiple approaches to allowing genericity in the transformation description object and converged on allowing the coercions to be generic themselves. This approach has the merit of being concise and extending naturally to any type constructor arity:

```
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 maintain 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 (up to the reference) when interleaved:

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

These semantics prevent the coercions from simply copying the value of the object into the new representation. For example, the referential equality condition above would be violated if the `toRepr` and `toHigh` methods would simply allocate new objects (which would get new references). Instead, the `toRepr` coercion would have to cache the original value so that, when decoding, the `toHigh` coercion could return the exact same object as originally given.

As expected, referential equality and mutability make transformations a lot more difficult. Luckily, in most use cases, the targets, such as library 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. A desirable extension of our approach would be to statically check the compatibility of the high-level type with its coercions. This could prevent the programmer from incorrectly copying internally mutable objects inside the coercions.

The complete transformation description object for the complex number encoding is given in the Appendix.

4.2 Transformation Scopes and Composability

Existing LDL-based data representation transformations, such as value class inlining and specialization, have fixed semantics and occur in separate compiler phases. Instead, the ADR transformation handles all 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 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 system to reference the description object, essentially referencing the transformation semantics with each individual value. We then leverage the type system and the signature persistence facilities to correctly transform all values, thus allowing scopes to safely and efficiently pass data among themselves, using the representation type—a property we refer to as 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 nested 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, the interested reader may consult the "Scope Nesting" page on the project wiki [4], which describes all cases of scope overlapping, collaboration and nesting. Furthermore, 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 follows from the scoped nature of the ADR transformation, which allows programmers to use the most efficient data representation for each task. But it raises the question of whether values can be safely passed across scopes that use different representations:

```
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. While type-checking, the description objects are stored as invisible abstract syntax tree attachments (described in §5):

```
var x: (Int, Int) = (3, 5) /* att: IntPairToLong */
val y: (Int, Int) = (2, 6) /* att: IntPairToDouble */
x = y
```

Then, during the INJECT phase, each value or method definition that matches the description object's high-level type is annotated with the `@repr` annotation, parameterized on the transformation description object:

```
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 COERCE 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 the description object `IntPairToLong`) while the right-hand side will become a floating point expression (based on the description object `IntPairToDouble`). However, both description objects have the same high-level type, the integer pair, which can be used as a middle ground in the conversion:

```
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:

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

In the end, 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 `adrt` scopes compose, allowing values to flow across different representations, from one scope to another. Similarly to the LDL transformation, the mechanism aims to employ a minimal number of conversions.

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

```
adrt(FloatPairAsLong) {
  adrt(IntPairAsLong) {
    val x: (Float, Float) = (1f, 0f)
    var y: (Int, Int) = (0, 1)
    // y = x
    // 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, thus introducing unsoundness.

The answer is no: the code is first type-checked against the high-level types even before the 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, informing 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, in some cases, would allow such unsound assignments.

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

```
adrt(PairAsMyPair) {     // (Int,Int) -> MyPair[Int,Int]
  adrt(IntPairAsLong) { // (Int,Int) -> Long
    val x: (Int, Int) = (2, 3)
  }
  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 `MyPair[Int, Int]` and `Long`. However, during the INJECT phase, if a value is matched by several nested `adrt` scopes, 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 (our current solution):

```
console:9: warning: Several adrt scopes can be applied
    to value x. Picking the innermost one: IntPairAsLong
val x: (Int, Int) = (2, 3)
       ^
```

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

A different case of nested transformation description objects is what we call "cascading" scopes:

```
adrt(TtoU) {       // T -> U
  adrt(UtoV) {     // U -> V
    val t: T = ???  // T -> U -> V (?)
  }
}
```

It may seem natural that the value `t` will be transformed to use the `V` representation type: first, converting from `T` to `U` and then from `U` to `V`. Unfortunately, the underlying mechanism, Late Data Layout [47], only allows values to undergo one representation change in the COERCE phase. Thus, to enable cascading scopes, we would have to either run the COERCE phase until a fixpoint or extend both the theory and the implementation to handle multiple conversions in a single run, neither of which is a straightforward extension. Therefore, in the current approach, we disallow cascading scopes:

```
cascading.scala:25: warning: Although you may expect
   value t to use the representation type U, by virtue
   of nesting the transformation description objects
   (TtoU,UtoV), "cascading" scopes are not supported:
val t: T = ???
       ^
```

Instead, the value `t` undergoes a single ADR transformation, to the representation type `V`. By disallowing "cascading" scopes we also protect against cyclic scopes, such as `TtoU` nested inside `UtoT`, which could cause infinite loops.

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.

To recap: 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 are annotated in exactly the same way. The next part extends scope composition across separate compilation.

4.3 Separate Compilation

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

```
adrt(IntPairComplexToLongComplex) {
 def gcd(n1: (Int,Int), n2: (Int,Int)): (Int,Int) =...
}
```

After the INJECT phase, the signature for method `gcd` is:

```
def gcd(
   n1: @repr(IntPairComplexToLongComplex) (Int, Int),
   n2: @repr(IntPairComplexToLongComplex) (Int, Int)
 ): @repr(IntPairComplexToLongComplex) (Int, Int) =
    ...
```

And, after the COMMIT phase executed, the bytecode signature for method `gcd` is:

```
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 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 `gcd`, a separate compilation could assume it accepts two pairs of integers as input. Yet, in the bytecode, the `gcd` method accepts long integers and cannot 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 `gcd` method. It would also decode the result of `gcd` back to a pair of integers. This approach allows calling `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 `gcd` method from a scope transformed using the same transformation description object as we used when compiling `gcd`, but in a different compilation run:

```
adrt(IntPairComplexToLongComplex) {
  val n1: (Int, Int) = ...
  val n2: (Int, Int) = ...
  val res: (Int, Int) = gcd(n1, n2)
}
```

In this case, would it make sense to call the bridge method? The values `n1` and `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, we want the `adrt` scopes to compose across separate compilation, allowing the call to go through in the encoded format. This is achieved by persisting the transformation information in the generated bytecode, but we have to do so without making ADR transformations a first-class concept. The approach we took is to persist the injected annotations, including the reference to the transformation description object. These become part of the signature of `gcd`:

```
// loaded signature (description object abbreviated):
def gcd(n1: @repr(.) (Int, Int), n2: @repr(.) (Int,
   Int)): @repr(.) (Int, Int)
```

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

```
val n1: @repr(.) (Int, Int) = ...
val n2: @repr(.) (Int, Int) = ...
val res: @repr(.) (Int, Int) = gcd(n1, n2)
```

With the complete signature for `gcd`, the COERCE phase does not introduce any coercions, since the arguments to method `gcd` use the same encoding as the method parameters did in the previous compilation run. This allows `adrt` scopes to seamlessly compose even across separate compilations. After the COMMIT phase, the scope is compiled to:

```
val n1: Long = ...
val n2: Long = ...
val res: Long = 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:

```
val res: (Int, Int) = gcd((55, 2), (17, 13))
```

Since the signature for method `gcd` references the transformation description object, the COERCE phase knows exactly which coercions are necessary:

```
val res: (Int, Int) = toHigh(...,
  gcd(toRepr(..., (55, 2)), toRepr(..., (17, 13))))
```

Generally, persisting references to the description objects in each value's signature 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 natural syntax and the 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:

```
adrt(IntPairComplexToLongComplex) {
  val n = (0, 1)
  println(n.toString)
}
```

When handling method calls on an encoded receiver, 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):

```
val n: Long = ...
println(IntPairComplexToLongComplex.toHigh(n).toString)
```

The price to pay is decoding the value into the high-level type, which usually leads to heap allocations and can introduce overheads. If a corresponding bypass method is available, the LDL transformation can use it:

```
val n: Long = ...
println(IntPairComplexToLongComplex.bypass_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 `bypass_toString` method in the transformation description object, and, if none is found, warns the programmer and proceeds with decoding the receiver and generating the dynamically-dispatched call.

Methods added via implicit conversions and other enrichment techniques, such as extension methods or type classes, add another layer or complexity, only handled in the ADR transformation. For example, we can see the multiplication operator `⋆`, added via an implicit conversion (we will further analyze the interaction with implicit conversions in §4.5):

```
adrt(IntPairComplexToLongComplex) {
  val n1 = (0, 1)
  val n2 = n1 ⋆ n1
}
```

Type-checking the program produces an explicit call for the implicit conversion that introduces the `⋆` operator:

```
val n1: (Int, Int) = (0, 1)
val n2: (Int, Int) = intPairAsComplex(n1) ⋆ n1
```

This is a costly pattern, requiring `n1` to be decoded into a pair and passed to the `intPairAsComplex` 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 combining the implicit method name and the operator. For simplicity, if we assume the name is `implicit_⋆` and the bypass exists in the `IntPairComplexToLongComplex` object, the COERCE phase transforms the code to:

```
val n1: Long = toRepr((0,1))
val n2: Long = IntPair...Complex.implicit_⋆(n1, n1)
```

This allows the call to the `⋆` operator to be transformed into a bypass call, avoiding heap object creation, and thus significantly improving the performance and heap footprint.

Bypass methods. Both normal and implicit bypass methods defined in the transformation description object need to correspond to the original method they are replacing and:

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

Therefore, during the COERCE phase, which introduces bypass method calls, the `implicit_⋆` has the signature:

```
def implicit_⋆(recv: @repr(...) (Int, Int), n2:
  @repr(...) (Int, Int)): @repr(...) (Int, Int)
```

Since the programmer defining the description object is free to choose any encoding for the bypass arguments, the following (suboptimal) signature would be equally accepted:

```
def implicit_⋆(recv:(Int,Int), n2:(Int,Int)):(Int,Int)
```

With the second signature, despite calling a bypass method, the arguments still have to be coerced, since the high-level type `(Int, Int)` is expected.

It is interesting to notice that representation-agnostic method rewriting relies on two previous design choices:
(1) shifting focus from scopes to individual values and
(2) carrying the entire transformation semantics in the signature of each encoded value. Yet, there is still a snag.

Constructors create heap objects before they can be encoded in the representation type. In our example, the first line runs the pair (`Tuple2`) constructor, which creates a heap object, and then converts it to the `Long` representation:

```
// In Scala, (0,1) is a shorthand for new Tuple2(0,1):
val n1: Long = toRepr((0,1))
val n2: Long = IntPair...Complex.implicit_⋆(n1, n1)
```

Instead of allocating the `Tuple2` object, the ADR transformation can intercept and rewrite constructor invocations into constructor bypass methods:

```
val n1: Long = IntPair...Complex.ctor_Tuple2(0, 1)
val n2: Long = IntPair...Complex.implicit_⋆(n1, n1)
```

Notice that the integers are now passed as arguments to the constructor bypass method `ctor_Tuple2`, by value. This completes this scope's transformation, allowing it to execute without allocating any heap object at all.

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 the additional steps that are taken to ensure correct program transformation.

Dynamic Dispatch and Overriding are 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:

```
trait X {
  def identity(i: (Int, Int)): (Int, Int) = i
}
adrt(IntPairAsLong) {
  class Y(t: (Int, Int)) extends X {
    override def identity(i: (Int, Int)) = t
  }
}
```

After the ADR transformation, the `identity` method in class `Y` no longer overrides method `identity` in trait `X`, since its signature expects a long integer instead of a pair of integers. To address this problem, we extend the Late Data Layout mechanism by introducing a new BRIDGE phase, which runs just before COERCE and inserts bridge methods to enable correct overriding. After the INJECT phase, the code corresponding to class `Y` is:

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

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

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

The COERCE 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`:

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

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 occurs when extending class `Y` from a scope with a different description:

```
adrt(IntPairAsDouble) { // != IntPairAsLong
  class Z extends Y(...) {
    override def identity(i: (Int, Int)): (Int, Int) = i
  }
}
```

The ensuing BRIDGE phase generates 2 bridge methods:

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

Although the resulting object layout is consistent, the `@bridge` methods have 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. In such cases the BRIDGE phase issues warnings to notify the programmer of a possible slowdown caused by the coercions.

Dynamic and Native Code. Thanks to the BRIDGE phase, class `Z` conforms to the trait (interface) `X`, thus, any call going through the interface will execute as expected, albeit, in this case, less efficiently. This allows dynamically loaded code to work correctly:

```
Class.forName("Z").newInstance() match {
  case x: X[_] => x.identity((3, 4))
  case _ => throw new Exception("...")
}
```

We have not tested the Java Native Interface (JNI) with ADR transformations, but expect the object layout assumptions in the C code to be invalidated. However, method calls should still occur as expected.

Generics. Another question 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:

```
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. 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:

```
def use1(list: List[(Int, Int)]): Unit = ...
def use2(list: List[(Int, Int)]): Unit = use1(list)
```

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

```
adrt(ListOfIntPairAsListOfLong) {
  def use3(list: List[(Int, Int)]): Unit = use1(list)
}
```

In this second example, following the COMMIT phase, the `List[(Int, Int)]` is indeed transformed to `List[Long]`:

```
def use3(list: List[Long]): Unit = use1(toHigh(list))
```

To summarize, `adrt` 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)`, outside generics

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

Implicit conversions interact in two ways with `adrt` scopes:

Extending the object functionality through implicit conversions, extension methods, or type classes must be taken into account by the method call rewriting in the COERCE 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 signature transformation rules remain the same (§4.4).

Offering an alternative transformation mechanism. Despite the apparent similarity, implicit conversions are not powerful enough to replace the ADRT mechanism. For example, assuming the presence of implicit methods to coerce integer pairs to longs and back, we can try to transform:

```
val n: (Int, Int) = (1, 0)
val a: Any = n
println(a)
```

To trigger the transformation, we update the type of `n` to `Long` in the source code and wait for the implicit conversions to do their work:

```
val n: Long = (1, 0) // triggers implicit conversion
val a: Any = n       // does not trigger the reverse
println(a)
```

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

With this, we presented the Ad hoc Data Representation Transformation mechanism and how it interacts with other language features to guarantee transformation correctness. The next section describes the architecture and implementation of our Scala compiler plugin.

5. Implementation

We implemented the ADR transformation as a Scala compiler plugin [3], by extending the open-source multi-stage programming transformation provided with the LDL [47] artifact, available at [5]. In this section we describe the technical aspects of our implementation that are not directly related to the transformation itself, but to providing a good programmer experience. Readers should also refer to the paper Appendix for an end-to-end example of the transformation phases. Additionally, the paper is accompanied by an artifact which can be used to explore the transformation.

The `adrt` 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:

```
def foo: (Int, Int) = {
  adrt(IntPairToLong) {
    val n: (Int, Int) = (2, 4)
  }
  n
}
```

Immediately after the source is parsed, the POSTPARSER phase transforms the `adrt` scopes in three steps:

- it attaches a unique id to each `adrt` scope;
- it records and clears the block enclosed by the `adrt` scope
- it inlines the recorded code immediately after the now-empty `adrt` scope and, in the process, it marks the value and method definitions by the `adrt` scope's unique id (or by multiple ids, if `adrt` scopes are nested).

Following the POSTPARSER phase, the code is:

```
def foo: (Int, Int) = {
  /* id: 100 */ adrt(IntPairToLong) {}
  /* id: 100 */ val n: (Int, Int) = (2, 4)
  n
}
```

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 `adrt` 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:

```
def foo: (Int, Int) = {
  val n: @repr(IntPairToLong) (Int, Int) = (2, 4)
  n
}
```

After the INJECT phase, the annotated signatures are persisted, allowing the scope composition to work across separate compilation. Later, the BRIDGE, COERCE and COMMIT 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 `toHigh` and `toRepr` coercions, which may be generic, as shown in §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 COERCE 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 COERCE phase, their signatures are expected to take parameters of the annotated high-level type, in order to allow redirecting method calls. To work around this problem, we added the `@high` annotation, which acts as an anti-`@repr` and marks the representation types:

```
object IntPairToLong extends TransformationDescription{
  ...
  // source-level signature (type-checking the body):
  def bypass_toString(repr: @high Long): String = ...
  // signature during coerce (allows rewriting calls):
  //  def bypass_toString(repr: @repr(...) (Int, Int))
  // signature after commit (bytecode signature):
  //  def bypass_toString(repr: Long)
}
```

This mechanism allows programmers to both define and use the transformation description objects in the same compilation run—an obvious benefit over full macro-based metaprogramming in Scala [19]. This reflects our design decision to only allow the description object to drive the transformation through its members and types, without running code that manipulates the AST.

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.

Compiler Entry Points. In many of the descriptions so far we have implicitly assumed the Scala compiler features. To ease other compiler developers in porting this approach, we highlight the exact Scala compiler features that we use:

- The type checker is available at all times during compilation;
- We can change/see a symbol's signature at any phase;
- The compiler supports type annotations and external annotation checkers;
- The compiler support AST attachments;
- The compiler offers expected type propagation during type checking (In Scala, this is part of the local type inference.)

This concludes the section, which explained how we solved the main technical problems in the ADR Transformation and how this impacted the compilation pipeline. We now continue with our experimental evaluation.

6. Benchmarks

This section evaluates the experimental benefits of ADR transformations in targeted micro-benchmarks and in the setting of a library and its clients.

We ran the benchmarks on an Intel `i7-4702HQ` quad-core processor machine with the frequency fixed at `2.2GHz`, and `2GB` of RAM, running the Oracle Java SE `1.7.0_80-b15` distribution on Ubuntu `14.04 LTS`. To avoid the noise caused by the just-in-time (JIT) compiler and garbage collection (GC) cycles, we measured the running times using the ScalaMeter benchmarking platform [36], which warms up the Java Virtual Machine according to statistically rigorous performance evaluation guidelines [26].

6.1 ADRT Micro-Benchmarks

Our benchmarking platform, ScalaMeter, executes micro-benchmarks using the following recipe:

- First, fork a new JVM;
- Execute the benchmark several times to warm up the JVM, only measuring the noise;
- When the noise drops below a threshold, execute the benchmark and gather measurements;

For each benchmark run, we monitor:

- The benchmark running time;
- GC cycles occurring during the run (in-benchmark);
- GC cycles occurring after the run (inter-benchmark);

At the end of a cycle, we manually trigger a full GC cycle so the current run does not affect the next. The memory collected after the run (inter-benchmark) corresponds to the input and output data and any garbage produced by running the benchmarked code that was not automatically collected during its execution (in-benchmark).

This allows us to record the following parameters for each benchmark:

- Benchmark running time (ms)
- In-benchmark garbage collected (MB)
- In-benchmark GC pause time (ms)
- Inter-benchmark garbage collected (MB)
- Inter-benchmark GC pause time (ms)

Since the ADR transformation is directly related to memory layout and, thus, to memory consumption, we paid special attention to GC cycles. Please notice that the benchmark running time includes the in-benchmark GC pause but not the inter-benchmark GC pause. This allows us to separately measure the speedups gained by avoiding GC cycles and from other factors, such as:

- Avoiding pointer dereferencing;
- Improving cache locality;
- Simplifying operations;
- Specializing operations;
- Lazyfying operations.

For each benchmark, we broke down the transformation in several steps, which allowed us to quantify the exact contri-

Benchmark	Time (ms)	Speedup	In-benchmark Garbage (MB)	In-benchmark GC time (ms)	Inter-benchmark Garbage (MB)	Inter-benchmark GC time (ms)
10K GCD runs, original	28.1	none	0	0	13.5	13
10K GCD runs, class	12.5	2.2x	0	0	2.5	10
10K GCD runs, boxed	15.0	1.9x	0	0	8.7	11
10K GCD runs, unboxed	2.2	12.7x	0	0	0.5	9

Table 1. Greatest Common Divisor benchmark results.

bution obtained by each transformation step. Unfortunately, due to space constraints, we cannot include the complete analysis in the paper. Interested readers can review it in the accompanying artifact or on the project website [4].

We chose representative micro-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 [50];
- averaging sensor readings + array of struct;
- computing the first 10000 Hamming numbers.

All benchmarks are fully automated and use the `adrt` markers and transformation description objects. 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, CPU-bound benchmark, where the main slowdown is caused by heap allocations and GC cycles. We broke down the transformation into four steps, with the result shown in Table 1. None of the transformations triggered GC pauses during the measured runs, but they did produce different amounts of garbage objects:

The "original" benchmark does not apply any transformation, thus modeling Gaussian integers using Scala's `Tuple2` class. Due to limitations in the specialization [21, 22] translation in Scala, the memory footprint of `Tuple2` classes is larger than it should be.

The "class" transformation applies an `adrt` transformation which encodes Gaussian integers as our own `Complex` class, essentially retrofitting specialization. This obtains a 2x speed improvement and reduces the garbage by 5x:

```
case class Complex(_1: Int, _2: Int)
```

The "boxed" transformation encodes Gaussian integers as long integers, but keeps them heap-allocated. This is slower than having our own class since it requires encoding values into the long integer representation. To achieve boxing, we use `java.lang.Long` objects, which the Scala backend does not unbox. The additional value encoding produces a small slowdown and for unknown reasons increases the garbage produced.

The "unboxed" transformation is the one shown throughout the paper. It encodes Gaussian integers as `scala.Long` values, which are automatically unboxed by the Scala compiler backend. This brings a significant speedup to the benchmark, allowing execution to occur without any heap allocation, as explained in §4.4. Compared to using pairs of integers, the speedup is almost 13x and the garbage is reduced by 27x.

The transformation description objects for the three transformations above range between 30 and 40 lines of code and include more operations than necessary for the benchmark, such as addition, multiplication, multiplication with integers, subtraction, etc.

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 can benefit from deforestation [50], which avoids the creation of intermediate collections after each `map` operation:

```
adrt(ListAsLazyList){
  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]`:

```
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 the collection 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.

To put the transformation into context, we explored several scenarios:

The "original" case executes the least squares method on 5 million points without any transformation. Table 2 shows

Benchmark	Time	Speedup	In-benchmark		Inter-benchmark	
			Garbage	**GC time**	**Garbage**	**GC time**
	(ms)		(MB)	(ms)	(MB)	(ms)
LSM, original	8264	none	1166	7547	809	5317
LSM, scala-blitz	3464	2.4x	468	2936	1165	5236
LSM, adrt generic	429	19.3x	701	3	933	5210
LSM, adrt miniboxed	280	29.5x	0	0	701	5193
LSM, manual deforestation	195	42.4x	0	0	702	5269
LSM, manual fusion	79	105.0x	0	0	702	5282

Table 2. Least Squares Method benchmark results.

that, on average, as much as 1.1 GB of heap memory is reclaimed during the benchmark run, significantly slowing down the execution. If it was not for the in-benchmark GC pause, the execution would take around 700ms, in line with the other transformations.

What we can also notice is that, across all benchmarks, the input data occupies around 700MB of heap space and is only collected at the end of the benchmark. A back-of-the-envelope calculation can confirm this: each linked list node takes 32 bytes (2-word header + 8-byte pointer to value + 8-byte pointer to the next cell) and contains a tuple of 48 bytes (2-word header + two 8-byte pointers and two 8-byte doubles, due to limitations in specialization), which itself contains 16 bytes per boxed double. Considering 5 million such nodes, we have: $(32+48+2\times16)*5\times10^6 = 560\times10^6$, approximately 560MB of data.

The "blitz" transformation uses the dedicated collection optimization tool `scalablitz` [8, 13] to improve performance. Under the hood, scalablitz uses compile-time macros to rewrite the code and improve its performance. Indeed, the tool manages to both cut down on garbage generation and improve the running performance of the code.

The "adrt" transformation performs deforestation by automatically introducing `LazyList`s. This avoids the creation of intermediate lists and thus significantly reduces the garbage produced. We tried using two versions of `LazyList`: one using erased generics (adrt generic) and one using miniboxing [46] specialization (adrt miniboxed).

The erased generic `LazyList` executed the code on par with the scalablitz optimizer but produced less garbage and the GC pause was much shorter (probably requiring a simple young-generation collection, not a full mark and sweep).

The miniboxed `LazyList`, on the other hand, both executed faster and did not produce any in-benchmark garbage. If we count in-benchmark GC pauses, the speedup produced by combining "adrt" scopes for deforestation and miniboxing for specialization is 29.5x compared to the original code. If we only count execution time, subtracting in-benchmark GC pauses, the speedup is 2.56x.

Manual transformations complete the picture: in the "deforestation" transformation we write C-like while loops by hand to traverse the input list. We use four separate loops, to simulate the best case scenario for an automated transformation. The result is a 1.43x speedup compared to "adrt miniboxed".

The "fusion" manual transformation unites the four separate input list traversals into a single traversal. While this transformation cannot be applied unless we assume a closed world, it is still interesting to compare our transformation to a best-case scenario. The manual fusion improves the performance by 3.54x compared to "adrt miniboxed". However, what we can notice is that both "adrt miniboxed" and the manual transformations produce the exact same amount of garbage: 700MB.

In terms of programmer effort, the `LazyList` definition takes about 60 LOC and the transformation description object about 30 LOC. The difference between "adrt erased" and "adrt miniboxed" is the presence of `@miniboxed` annotations in the `LazyList` classes and in the description object.

The Sensor Readings benchmark is inspired by the Sparkle visualization tool [10], which 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 Sparkle processing core and automated the array-of-struct to struct-of-array transform:

```
type SensorReadings = Array[(Long, Long, Double)]
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 = ...
 ...
}
```

In the benchmark, we have an array of 5 million events, each with its own timestamp, type and reading. We are seeking to average the readings of a single type of event occurring in the event array. Since our transformation influences cache locality, we had two different speedups depending on the event distribution:

- Randomly occurring events are triggered with a probability of 1/3 in the sensor reading array;

Benchmark	Time (ms)	Speedup	In-benchmark Garbage (MB)	In-benchmark GC time (ms)	Inter-benchmark Garbage (MB)	Inter-benchmark GC time (ms)
array of struct, random	55.5	none	0	0	451	15
struct of array, random	30.4	1.8x	0	0	435	13
array of struct, uniform	32.5	none	0	0	454	16
struct of array, uniform	5.7	5.7x	0	0	433	19
10001-th number, original	6.56	none	0	0	31	11
10001-th number, step 1	2.70	2.4x	0	0	31	11
10001-th number, step 2	2.16	3.0x	0	0	31	12
10001-th number, step 3	1.64	4.0x	0	0	31	10

Table 3. Sensor Readings and Hamming Numbers benchmark results.

- Uniformly occurring events appear every 3rd element, thus offering more room for CPU speculation.

Using the `adrt` scope to transform the array of tuples into a tuple of arrays allows better cache locality and fewer pointer dereferences. With random events, the "adrt" transformation produces a speedup of 1.8x. With uniformly distributed events, both the original and the transformed code run faster, yet resulting in a speedup of 5.7x.

In all four cases, the amount of memory allocated is approximately the same and no objects are allocated aside from the input data. Thus, the operation speedups are obtained through improving cache locality.

The transformation description object is 50 LOC and requires 20 additional LOC to define implicit conversions.

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 [7] and attempted to speed up:

```scala
adrt(BigIntToLong) {
 adrt(QueueOfBigIntAsFunnyQueue) {
   class Hamming extends Iterator[BigInt] {
     import scala.collection.mutable.Queue
     val q2 = new Queue[BigInt]
     val q3 = new Queue[BigInt]
     val q5 = new Queue[BigInt]
     def enqueue(n: BigInt) = {
       q2 enqueue n * 2
       q3 enqueue n * 3
       q5 enqueue n * 5
     }
     def next = {
       val n = q2.head min q3.head min q5.head
       if (q2.head == n) q2.dequeue
       if (q3.head == n) q3.dequeue
       if (q5.head == n) q5.dequeue
       enqueue(n); n
     }
     def hasNext = true
     q2 enqueue 1
     q3 enqueue 1
     q5 enqueue 1
   }
 }
}
```

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[BigIng]` by a fixed-size circular buffer built on an array. The result was an 4x 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 good enough. This is similar to what a dynamic language virtual machine would do: it would make assumptions based on the code and would automatically de-specialize the code if the assumption is invalidated. In our case, when the assumption is invalidated, the code will throw an exception.

As with other benchmarks, we broke down the transformation is several steps:

The "original" code is the unmodified version from the Rosetta Code website, which we kept as a witness.

The "step1" code uses `adrt` scopes to replace the `Queue` object with a custom, fixed-size array-based circular buffer. This collection specialization brings a 2.4x speedup without any memory layout transformation.

The "step2" code uses `adrt` scopes to replace the `BigInt` object in both class `Hamming` and the circular buffer by boxed `java.lang.Long` objects. This additional range restriction brings an extra 1.25x speedup.

The "step3" code replaces the `BigInt` objects by unboxed `scala.Long` values. This unboxing operation produces an additional 1.31x speedup, as fewer objects are created during the benchmark execution.

The conclusion is that, although the ADR transformation can be viewed as a memory layout optimization, it can additionally trigger more optimizations that bring orthogonal speedups, such as specializing operations and collections.

For this example, the two transformation objects are 100 LOC and the circular buffer is another 20 LOC.

6.2 ADRT in Realistic Libraries

The `adrt` scoped transformation is a conceptual generalization of a mechanism motivated by library transformation scenarios. In particular, the resulting data representation transformation is used in conjunction with the miniboxing transformation [6, 46], in order to replace standard library

Benchmark	Generic	Miniboxed	Miniboxed +functions
Sum	98.2 ms	158.6 ms	18.0 ms
SumOfSquares	131.6 ms	193.1 ms	12.0 ms
SumOfSqEven	92.3 ms	189.6 ms	48.7 ms
Cart	217.4 ms	214.9 ms	57.5 ms

Table 4. Scala Streams pipelines for 10M elements.

functions and *tuples* by custom, optimized versions adequate for miniboxed code [48]. The scope of this data representation transformation is miniboxing-transformed code.

The miniboxing transformation [46] 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 [21], 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 [25] 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 miniboxed code incurs significant overhead.

Consider, for example, functions. (Tuples raise similar issues.) Scala 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) => x + 1`:

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

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 miniboxed code that uses functions, they have two choices: either accept the slowdown caused by converting the representation or define their own miniboxed `Function1` class, and perform the function desugaring by hand. Neither of these is a good solution.

Our data representation transformation converts the references to `Function1` in miniboxed code to the optimized `MiniboxedFunction1`, which allows calls to use the miniboxed representation, thus being more efficient. The problem is that the miniboxed code needs to interoperate with library-defined code, or with other libraries that were not transformed. Thus, the miniboxed code acts as a scope for the *function and tuple representation transformation*, i.e., the ADR transformation of `Function` and `Tuple`. This transformation has a significant impact in library benchmarks.

Benchmark	Running time
Manual C-like code	0.650 μs
Miniboxing with functions	0.705 μs
Miniboxing without functions	3.080 μs
Generic	13.409 μs

Table 5. Mapping a 1K vector.

The Scala-Streams library [14] imitates the design of the Java 8 stream library, to achieve high performance (relative to standard Scala libraries) for functional operations on data streams. The library 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.

Table 4 shows the result of applying our data representation transformation to the Scala-Streams published benchmarks. (The benchmarks are described in detail in prior literature [14, 15].) As can be seen, the miniboxing transformation is an enabler of our optimization but produces *worse* results by itself (due to extra conversions).

Compared to the original library, the application of miniboxing and data representation optimization for functions achieves a very high speedup—up to 11x for the SumOfSquares benchmark. In fact, the speedup relative to the miniboxed code without the function representation optimization is nearly 16x!

The Framian Vector implementation is an exploration into deeply specializing the immutable `Vector` bulk storage without using reified types [11, 12]. This is a benchmark created by a third party (a commercial entity using Scala). Table 5 shows a 4.4x speed improvement when the function representation is optimized and shows that the ADR-transformed function code performs within 10% of the fully specialized and manually optimized code.

7. Related Work

Changing data representations is a well-established and time-honored programming need. Techniques for removing abstraction barriers have appeared 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.

As discussed earlier, the standard optimizations that are closest to our approach are value classes [9] and class specialization [21, 46]. 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 [27]. Rose has an analogous proposal for value classes [38, 39] 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 [23] or inlining and escape analysis [20, 41]. Furthermore, modern VMs, such as V8, Truffle [52] and PyPy [16] 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 [47] mechanism in the Scala setting. Similar approaches, with different specifics in the extent of type system and customization support, have been applied to Haskell [29]. Foundational work exists for ML, with Leroy [32] presenting a transformation for unboxing objects, with the help of the type system. Later work extends [44] and generalizes [40] such transformations. In terms of runtime-dispatched generics, we refer to the work on Napier88 [34] and the TIL compiler [28, 43].

In the specific setting of data structure specialization, the CoCo approach [53] 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 [17], for instance, switches representations once collections become polymorphic or once they acquire many elements. The Scala Blitz optimizer uses macros to improve collection performance [8, 13].

Among mechanisms for extending an interface, such as extension methods, implicit conversions [35] and type classes [51] we can also mention views, which allow data abstraction and extraction through pattern matching [49].

Multi-stage programming [42] is another technique that optimizes the data representation. Its Scala implementation, dubbed lightweight modular staging, can both optimize and even re-target parts of a program to GPUs [18, 37]. 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 [30], based on Scala macros [19]. Another type-directed transformation in the Scala compiler is the pickling framework [33], also based on macros. Instead of transforming the data representation in-place, pickler combinators create serialization code that 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 1.8x and 24.5x. We demonstrated our mechanism in the Scala language, yet speculate that the same principles are applicable in different language settings.

Acknowledgements

We would like to thank Alexandru Nedelcu, whose comments inspired the initial work on miniboxed functions. The ADRT mechanism extends and generalizes the approach taken to introduce miniboxed functions in high-performance libraries. We are grateful to the members of the LAMP laboratory in EPFL for their extraordinary support, especially to Dmitry Petrashko, Heather Miller, Manohar Jonnalagedda and Sandro Stucki for their constructive feedback which shaped the project. We would like to thank Tom Switzer and Nicolas Stucki for building their applications on the miniboxing plugin and reporting the issues they came across. Vera Salvisberg and the OOPSLA reviewers provided helpful suggestions on improving the paper.

References

[1] Scala Streams Repository. URL https://web.archive.org/web/20150719095701/https://github.com/biboudis/scala-streams.

[2] Gaussian integers wikipedia article. URL https://web.archive.org/web/20150627074634/https://en.wikipedia.org/wiki/Gaussian_integer.

[3] ILDL Scala Compiler Plugin Source Code (also available in the artifact attached to the paper), . URL https://web.archive.org/web/20150719095959/https://github.com/miniboxing/ildl-plugin.

[4] ILDL Scala Compiler Plugin Documentation (also available in the artifact attached to the paper), . URL https://web.archive.org/web/20150719095927/https://github.com/miniboxing/ildl-plugin/wiki.

[5] LDL-based Staging Scala Compiler Plugin. URL https://web.archive.org/web/20150719095846/https://github.com/miniboxing/staging-plugin.

[6] The Miniboxing plugin website. URL https://web.archive.org/web/20141218011004/http://scala-miniboxing.org/.

[7] Rosetta Code Website. URL https://web.archive.org/web/20090106202220/http://rosettacode.org/.

[8] ScalaBlitz Optimizer. URL https://web.archive.org/web/20141218200210/http://scala-blitz.github.io/.

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

[10] Sparkle Tool. URL https://web.archive.org/web/20150719095739/https://github.com/mighdoll/sparkle.

[11] Optimistic Respecialization Attempts 1-5, . URL https://web.archive.org/web/20150719095636/http://io.pellucid.com/blog/optimistic-respecialization.

[12] Optimistic Respecialization Attempt 6, . URL https://web.archive.org/web/20150719095603/http://io.pellucid.com/blog/optimistic-respecialization-attempt-6.

[13] P. Aleksandar, D. Petrashko, and M. Odersky. Efficient Lock-Free Work-stealing Iterators for Data-Parallel Collections. In *PDP '15*. IEEE, 2015.

[14] A. Biboudis, N. Palladinos, and Y. Smaragdakis. Clash of the Lambdas. ICOOLPS, 2014.

[15] A. Biboudis, N. Palladinos, G. Fourtounis, and Y. Smaragdakis. Streams a la carte: Extensible Pipelines with Object Algebras. In *ECOOP '15*. ACM, 2015.

[16] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOLPS*, Genova, Italy, 2009. ACM.

[17] C. F. Bolz, L. Diekmann, and L. Tratt. Storage Strategies for Collections in Dynamically Typed Languages. In *OOPSLA*, OOPSLA '13. ACM, 2013.

[18] 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.

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

[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] A. Gal. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *PLDI*. ACM, 2009.

[24] C. F. Gauss. *Theoria residuorum biquadraticorum*, volume 1. Apud Dieterich, 1828.

[25] 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/`.

[26] A. Georges, D. Buytaert, and L. Eeckhout. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '07, 2007.

[27] B. Goetz. State of the Specialization, 2014. URL `https://web.archive.org/web/20150102143158/http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html`.

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

[29] 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.

[30] V. Jovanovic, V. Nikolaev, N. D. Pham, V. Ureche, S. Stucki, C. Koch, and M. Odersky. Yin-Yang: Transparent Deep Embedding of DSLs. Technical report, EPFL, 2013.

[31] B. W. Lampson and H. E. Sturgis. Reflections on an Operating System Design. *Commun. ACM*, 1976. ISSN 0001-0782.

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

[33] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant Pickles: Generating Object-oriented Pickler Combinators for Fast and Extensible Serialization. In *OOPSLA*. ACM, 2013.

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

[35] B. C. Oliveira, T. Schrijvers, W. Choi, W. Lee, and K. Yi. The Implicit Calculus: A New Foundation for Generic Programming. In *PLDI*. ACM, 2012.

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

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

[38] J. Rose. Value Types and Struct Tearing , . URL `https://web.archive.org/web/20150530182530/https://blogs.oracle.com/jrose/entry/value_types_and_struct_tearing`.

[39] J. Rose. Value Types in the VM, . URL `https://web.archive.org/web/20150525232233/https://blogs.oracle.com/jrose/entry/value_types_in_the_vm`.

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

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

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

[43] 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.

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

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

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

[47] V. Ureche, E. Burmako, and M. Odersky. Late Data Layout: Unifying Data Representation Transformations. In *OOPSLA '14*. ACM, 2014.

[48] V. Ureche, M. Stojanovic, R. Beguet, N. Stucki, and M. Odersky. Improving the Interoperation between Generics Translations. In *PPPJ*. ACM, 2015.

[49] P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *PoPL*. ACM, 1987.

[50] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP '88*. North-Holland Publishing Co., 1988.

[51] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *PoPL*. ACM, 1989.

[52] 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.

[53] G. Xu. CoCo: Sound and Adaptive Replacement of Java Collections. In *ECOOP*. Springer-Verlag, 2013.

Appendix

In this appendix we give the signatures of the Gaussian Integer transformation object and walk through each step of the compilation. The complete source code can be found in the artifact distributed with the paper:

```
package ildl.benchmark.gcd_minimal
import ildl._

object IntPairAsGaussianInt extends Transformation{
  // coercions:
  def toRepr(pair: (Int, Int)): @high Long = ...
  def toHigh(l: @high Long): (Int, Int) = ...

  // constructor:
  def ctor_Tuple2(_1: Int, _2: Int): @high Long = ...

  // interface:
  def implicit_GaussianInt_%(n1: @high Long, n2: @high
     Long): @high Long = ...
  def implicit_GaussianInt_norm(n: @high Long): Int =
     ...
}

object GCD {
  implicit class GaussianInt(pair: (Int, Int)) {
    def %(that: (Int, Int)): (Int, Int) = ...
    def norm = ...
  }

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

The most important compiler phases injected by the ADRT plugin are: POSTPARSER, INJECT, BRIDGE, COERCE and COMMIT. We show how each of these phases transforms the code. After the source code has been parsed, before type checking and name resolution, the POSTPARSER phase inlines the `adrt` scopes and attaches unique ids to the abstract syntax tree (AST) nodes, both for the transformation object and for the transformed scope:

```
object IntPairAsGaussianInt extends Transformation{
  // same as before
}
```

```
object GCD {
  // The GaussianInt class does not change:
  implicit class GaussianInt(pair: (Int, Int))...

  /* id: 100 */ adrt(IntPairAsGaussianInt) {}
  /* id: 100 */ def gcd(...): (Int, Int) = {
  /* id: 100 */   val remainder = n1 % n2
  /* id: 100 */   if (remainder.norm == 0) ...
  /* id: 100 */ }
}
```

After the POSTPARSER phase, the tree is ready for name resolution and type checking. These two phases run in tandem and transform the literal `IntPairAsGaussianInt` into a fully qualified reference, which points to the object symbol. Along the way, the type-checker ensures that `IntPairAsGaussianInt` extends the `Transformation` trait and that it is an object.

During type checking, the missing type annotations and implicit conversions are added to the AST:

```
object GCD {
  ...
  /* id: 100 */ adrt(IntPairAsGaussianInt) {}
  /* id: 100 */ def gcd(...): (Int, Int) = {
  /* id: 100 */   val remainder: (Int, Int) =
      new GaussianInt(n1).%(n2)
  /* id: 100 */   if ((new GaussianInt(remainder).norm)
      == 0) ...
  /* id: 100 */ }
}
```

After name resolution and type checking, the INJECT phase transforms the tree attachments into annotations. Since there is a single transformation object in the example, we will use `@repr` to mean `@repr(IntPairAsGaussianInt)`:

```
object GCD {
  ...
  def gcd(n1: @repr (Int, Int), n2: @repr (Int, Int)):
     @repr (Int, Int) = {
    val remainder: @repr (Int, Int) = ...
    if ((new GaussianInt(remainder).norm) == 0) ...
  }
}
```

The INJECT phase takes place right before the Scala signatures are persisted. Therefore, it needs to change the signatures in the `IntPairAsGaussianInt` object as well, by replacing all references to `@high Long` by `@repr (Int, Int)`, except for the two coercions:

```
object IntPairAsGaussianInt extends Transformation{
  // coercions:
  def toRepr(pair: (Int, Int)): @high Long = ...
  def toHigh(l: @high Long): (Int, Int) = ...

  // constructor:
  def ctor_Tuple2(_1: Int, _2: Int): @repr (Int, Int)

  // and so on ...
}
```

The member signatures are then persisted, meaning that all future compilation runs see the signatures left by the INJECT phase. Thus, to ensure scope composition, none of the signatures computed by the INJECT phase can contain references to the representation type, except for the `toHigh` and `toRepr` coercions. Then, all signatures that are transformed contain two pieces of information: the high-level type and the transformation description object.

As explained in §4.3 and §4.5, bridges are only necessary when a transformed method overrides or implements a method that was not transformed, in order to preserve the object model despite the low-level signature change. In our case, the `gcd` method neither implements existing interfaces nor overrides existing methods. Thus, the BRIDGE phase leaves the AST unchanged. Should the `gcd` method be called from outside an `adrt` scope, the arguments and return are adapted at the call site, based on the `@repr` annotation, which is persisted in method `gcd`'s signature.

Then, the COERCE phase introduces coercions and rewrites dynamic calls to bypass methods. The transformation description objects are skipped by the COERCE phase, as re-type-checking them with the modified signatures would lead to errors:

```
object GCD {
  ...
  def gcd(n1: @repr (Int, Int), n2: @repr (Int, Int)):
     @repr (Int, Int) = {
    val remainder: @repr (Int, Int) =
      implicit_GaussianInt_%(n1, n2)
    if (implicit_GaussianInt_norm(remainder) == 0) n2
      else gcd(n2, remainder)
  }
}
```

Finally, the COMMIT phase transforms the code to:

```
object IntPairAsGaussianInt extends Transformation{
  // coercions:
  def toRepr(pair: (Int, Int)): Long = ...
  def toHigh(l: Long): (Int, Int) = ...

  // and so on ...
}

object GCD {
  ...
  def gcd(n1: Long, n2: Long): Long = {
    val remainder = implicit_GaussianInt_%(n1, n2)
    if (implicit_GaussianInt_norm(remainder) == 0) n2
      else gcd(n2, remainder)
  }
}
```

Later in the compilation pipeline, the `Long` integer is unboxed to `long`, producing the following bytecode (for which we give the source-equivalent Scala code):

```
object IntPairAsGaussianInt extends Transformation{
  // coercions:
  def toRepr(pair: (Int, Int)): long = ...
  def toHigh(l: long): (Int, Int) = ...

  // and so on ...
}

object GCD {
  ...
  def gcd(n1: long, n2: long): long = {
    val remainder = implicit_GaussianInt_%(n1, n2)
    if (implicit_GaussianInt_norm(remainder) == 0) n2
      else gcd(n2, remainder)
  }
}
```

This is the bytecode that will ultimately execute in the Java Virtual Machine. Notice the fact that it executes without any object allocation and does not use dynamic dispatch. This ensures good performance and minimizes the garbage collection pauses.