FIRST LINE OF TITLE
SECOND LINE OF TITLE

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.

Thèse n. 1234 2011
présenté le 12 Mars 2011
à la Faculté des Sciences de Base
laboratoire SuperScience
programme doctoral en SuperScience
École Polytechnique Fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Paolino Paperino

acceptée sur proposition du jury:

Prof Name Surname, président du jury
Prof Name Surname, directeur de thèse
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur

Lausanne, EPFL, 2011

Wings are a constraint that makes
it possible to fly.
— Robert Bringhurst

To my parents. . .

Acknowledgements

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Lausanne, 12 Mars 2011 D. K.

Preface

A preface is not mandatory. It would typically be written by some other person (eg your thesis director).

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Lausanne, 12 Mars 2011 T. D.

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Key words:

Zusammenfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Stichwörter:

Résumé

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Mots clefs :

Contents

List of Figures

List of Tables

Introduction

A non-numbered chapter...

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

1 Miniboxing

1.1 Introduction

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

But despite the uniformity exposed to programmers, the lower level translation of generic code struggles with fundamentally non-uniform data. To illustrate the problem, we can analyze the `contains` method of a linked list parameterized on the element type, `T`, written in the Scala programming language:

```
1 def contains(element: T): Boolean = ...
```

When translating the `contains` method to lower level code, such as *assembly* or *bytecode* targeting a *virtual machine*, a compiler needs to know the exact type of the parameter, so it can be correctly retrieved from the stack, registers or read from memory. But since the list is generic, the type parameter `T` can have different bindings, depending on the context, ranging from a byte to a floating point number or a pointer to a heap object, each with different sizes and semantics. So the compiler needs to bridge the gap between the uniform interface and the non-uniform low level implementation.

Two main approaches to compiling generic code are in use today: heterogeneous and homogeneous. *Heterogeneous translation* duplicates and adapts the body of a method for each possible type of the incoming argument, thus producing new code for each type used. On the other hand, *homogeneous translation*, typically done with *erasure*, generates a single method but requires data to have a common representation, irrespective of its type. This

common representation is usually chosen to be a heap object passed by reference, which leads to indirect access to values and wasteful data representation. This, in turn, slows down the program execution and increases heap requirements. The conversions between value types and heap objects are known as *boxing* and *unboxing*. A different uniform representation, typically reserved to virtual machines for dynamically typed languages, uses the *fixnum* [**?**] representation. This representation can encode different types in the same unit of memory by reserving several bits to record the type and using the rest to store the value. Aside from reducing value ranges, this representation also introduces delays when *dispatching* operations, as the value and type need to be unpacked. An alternative is the *tagged union* representation [**?**], which does not restrict the value range but requires more heap space.

C++ [**?**] and the .NET Common Language Runtime [**?**, **?**] have shown that on-demand heterogeneous translations can obtain good performance without generating significant amounts of low level code. However, this comes at a high price: C++ has taken the approach of on-demand compile-time template expansion, where compiling the use of a generic class involves instantiating the template, type checking it and generating the resulting code. This provides the best performance possible, as the instantiated template code is monomorphic, but undermines separate compilation in two ways: first, libraries need to carry source code, namely the templates themselves, to allow separate compilation, and second, multiple instantiations of the same class for the same type arguments can be created during different compilation runs, and need to be eliminated in a later linking phase. The .NET Common Language Runtime takes a load-time, on-demand approach: it compiles generics down to bytecode with type information embedded, which the virtual machine specializes, at load-time, for the type arguments. This provides good performance at the expense of more a complex virtual machine and lock-step advancements of the type system and the virtual machine implementation.

In trying to keep separate compilation and virtual machine backward compatibility, the Java programming language [**?**] and other statically typed JVM languages [**?**, **?**, **?**, **?**] use homogeneous translations, which sacrifice performance. Recognizing the need for execution speed, Scala *specialization* [**?**] allows an *annotation-driven*, *compatible* and *opportunistic* heterogeneous transformation to Java bytecode. Programmers can explicitly annotate generic code to be transformed using a heterogeneous translation, while the rest of the code is translated using boxing [**?**]. Specialization is a compatible transformation, in that specialized and homogeneously translated bytecode can be freely mixed. For example, if both a generic call site and its generic callee are specialized, the call will use primitive values instead of boxing. But if either one is not specialized, the call will fall back to using boxed values. Specialization is also opportunistic in the way it injects specialized code into homogeneous one. Finally, being annotation-driven, it lets programmers decide on the tradeoff between speed and code size.

Unfortunately the interplay between separate compilation and compatibility forces specialization to generate all *heterogeneous variants* of the code during the class compilation instead of delaying their instantiation to the time they are used, like C++ does. Although in some libraries this behavior is desirable [**?**], generating all heterogeneous variants up front means

specializing must be done cautiously so the size of the generated bytecode does not explode. To give a sense of the amount of bytecode produced by specialization, for the Scala programming language, which has 9 primitive value types and 1 reference type, fully specializing a class like `Tuple3` given below produces 10^3 classes, the Cartesian product of 10 variants per type parameter:

```
1 class Tuple3[A, B, C](a: A, b: B, c: C)
```

In this paper we propose an alternative translation, called *miniboxing*, which relies on a very simple insight to reduce the bytecode size by orders of magnitude: since larger value types (such as integers) can hold smaller value types (such as bytes), it is enough for a heterogeneous translation to generate variants for the larger value types. In our case, on the Java Virtual Machine, miniboxing reduces the number of code variants from 10 per type parameter to just 2: reference types and the largest value type in the language, the long integer. In the `Tuple3` example, miniboxing only generates 2^3 specialized variants, two orders of magnitude less bytecode than specialization. Miniboxed code is faster than homogeneous code, as data access is done directly instead of using boxing. Unlike fixnums and tagged unions, miniboxing does not attach the type information to values but to classes and methods and thus leverages the language's static type system to optimize storage. Furthermore, the full miniboxing transformation eliminates the overhead of dispatching operations by using load-time class cloning and specialization (§1.6). In this context, our paper makes the following contributions:

- Presents an encoding that reduces the number of variants per type parameter in heterogeneous translations (§1.3) and the code transformations necessary to use this encoding (§1.4);
- Optimizes bulk storage (arrays) in order to reduce the heap footprint and maintain compatibility to homogeneous code, produced using erasure (§1.5);
- Utilizes a load-time class transformation mechanism to eliminate the cost of dispatching operations on encoded values (§1.6).

The miniboxing encoding can reduce duplication in any heterogeneous translation, as long as the following criteria are met:

- The value types of the statically typed target language can be encoded into one or more larger value types (which we call *storage types*) - in the work presented here we use the long integer as the single storage type for all of Scala's primitive value types;
- Conversions between the value types and their storage type do not carry significant overhead (no-op conversions are preferable, but not required);

- The set of operations allowed on generic values in the language is fixed (similar to fixing the where clauses in PolyJ [**?**]);
- All value types have boxed representations, in order to have a common data representation between homogeneous and miniboxed code. This representation is used to ensure compatibility between the two translations.

In order to optimize the code output by the miniboxing transformation, this paper explores the interaction between value encoding and array optimization on the HotSpot Java Virtual Machine. The final miniboxing transformation, implemented as a Scala compiler plug-in[1], approaches the performance of monomorphic code, matches the performance of specialization, and obtains speedups of up to 22x over the current homogeneous translation, all with modest increases in bytecode size (§1.7).

The paper will first explain the specialization transformation (§1.2) upon which miniboxing is built. It will then go on to explain the miniboxing encoding (§1.3), transformation (§1.4), runtime support (§1.5) and load-time specialization (§1.6). It will finish by presenting the evaluation (§1.7), surveying the related work (§1.8) and concluding (§1.9).

1.2 Specialization in Scala

This section presents specialization [**?**], a heterogeneous translation for parametric polymorphism in Scala. Miniboxing builds upon specialization, inheriting its main mechanisms. Therefore a good understanding of specialization and its limitations is necessary to motivate and develop the miniboxing encoding (§1.3) and transformation (§1.4).

There are two major approaches to translating parametric polymorphism to Java bytecode: homogeneous, which requires a common representation for all values, and heterogeneous, which duplicates and adapts code for each type. By default, both the Scala and Java compilers use homogeneous translation with each value type having a corresponding reference type. Boxing and unboxing operations jump from one representation to the other. For example, `int` has `java.lang.Integer` as its corresponding reference type.

Boxing enables a uniform low level data representation, where all generic type parameters are translated to references. While this simplifies the translation to bytecode, it does come with several disadvantages:

- Initialization cost: allocating an object, initializing it and returning a pointer takes longer than simply writing to a processor register;
- Indirect access: Extracting the value from a boxed type requires computing a memory address and accessing it instead of simply reading a processor register;

[1] Available at http://scala-miniboxing.org/.

- Undermined data locality: Seemingly contiguous memory storages, such as arrays of integers, become arrays of pointers to heap objects, which may not necessarily be aligned in the memory. This can affect cache locality and therefore slow down the execution;
- Heap cost: the boxed object lives on the heap until it is not referenced anymore and is garbage collected. This puts pressure on the heap and triggers garbage collection more often.

To eliminate the overhead of boxing, the Scala compiler features specialization: an annotation-driven, compatible and opportunistic heterogeneous transformation. Specialization is based on the premise that not all code is worth duplicating and adapting: code that rarely gets executed or has little interaction with value types is better suited for homogeneous translation. Since a compile-time transformation such as specialization has no means of knowing how code will be used, it relies on programmers to annotate which code to transform. Recent research in JavaScript interpreters [**?**, **?**] uses profiling as another method of triggering compatible specialization of important traces in the program.

With specialization, programmers explicitly annotate the code to be transformed heterogeneously (§1.2.1 and §1.2.2) and the rest of the program undergoes homogeneous translation. The bytecode generated by the two translations is compatible and can be freely mixed. This allows specialization to have an opportunistic nature: it injects specialized code, in the form of specialized class instantiations and specialized method calls (§1.2.3), but the injected entities are always compatible with the homogeneous translation (§1.2.4). However, the interaction with separate compilation leads to certain limitations that miniboxing addresses (§1.2.5).

1.2.1 Class Specialization

To explain how specialization applies the heterogeneous translation, we can use an immutable linked list example:

```
1 class ListNode[@specialized T]
2         (val head: T, val tail: ListNode[T]) {
3   def contains(element: T): Boolean = ...
4 }
```

Each `ListNode` instance stores an element of type `T` and a reference to the tail of the list. The `null` pointer, placed as the tail of a list, marks its end. A real linked list from the Scala standard library is more sophisticated [**?**, **?**], but for the purpose of describing specialization this example is sufficient. It is also part of the benchmarks presented in the Evaluation section (§1.7), as it depicts the behavior of non-contiguous collections that require random heap access.

The `ListNode` class has the generic `head` field, which needs to be specialized in order to avoid boxing. To this end, specialization will duplicate the class itself and adapt its fields for each

primitive value type. Figure 1.1 shows the class hierarchy created: the parent class is the homogeneous translation of `ListNode`, which we also call generic class. The 10 subclasses are the specialized variants. They correspond to the 8 Java primitive types, `Unit` (which is Scala's object-oriented representation of `void`) and reference types[2]. Each of these specialized classes contains a `head` field of a primitive type, and inherits (or overrides) methods defined in the generic class. So far, specialization duplicated the class and adapted the fields, but in order to remove boxing the methods also need to be transformed heterogeneously.

1.2.2 Method Specialization

In the specialized variants of `ListNode`, the `contains` method needs to be duplicated and adapted to accept primitive values as arguments instead of their boxed representations. Since the `contains` method is already inherited from the generic class, it actually needs to be overridden. But it cannot be overridden, because its signature after the erasure [**?**] transformation expects a reference type (`java.lang.Object`) and the specialized signature expects a primitive value. Therefore specialized methods need to be name-mangled, giving birth to new methods such as `contains_I` for `Int` and `contains_J` for `Long`.

The `contains` method from the generic parent class will be inherited by all the specialized classes. But its code is generic and does not make use of primitive values, which is suboptimal. Therefore each specialized class overrides the generic `contains` and redirects it to the corresponding specialized variant, such as `contains_I` or `contains_J`. The redirection is done by unboxing the argument received by `contains` and calling the specialized method with the value type, as shown in Figure 1.2. The same transformation is applied for accessors of specialized fields, such as `head` in the `ListNode` class.

1.2.3 Opportunistic Tree Transformation

The program code can only refer to generic classes and methods, not their specialized variants. This happens because the specialization phase, which creates the variants, runs after the type checking phase. Thus the program is checked only against the generic classes and methods. But this does not mean specialization duplicates code in vain: aside from creating the variants, specialization also injects the specialized variants in the program code.

The last step in eliminating boxing is rewriting the Scala abstract syntax tree (AST) to instantiate specialized classes and use specialized methods. We call this process rewiring. Rewiring works across separate compilation, as the specialization metadata is written in the generated bytecode. This makes is possible to use specialized code from libraries.

The instantiation rewiring injects specialized classes when the **new** keyword is used. When the instantiated class has a more specific specialized variant for the given type arguments,

[2] Technical note: For a single type parameter the reference variant will not be generated and the generic class will be used instead.

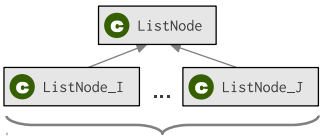


Figure 1.1 – Class hierarchy generated by Specialization. The letters in class suffix represent the type they are specialized for: V-Scala Unit, Z-Boolean, B-Byte ... J-Long, L-AnyRef. The names are simplified throughout the paper, and we avoid discussing the problem of name mangling, which was addressed in [**?**].

the instantiation is rewired. Despite constructing a different class, the types in the AST are not adjusted to reflect this: In the example given below, although the instantiation is rewired to `new ListNode_I`, the type of `node1` remains `ListNode[Int]`. This makes specialization compatible: whether or not the instantiation is rewired, both the specialized class and the generic class are still subtypes of `ListNode[Int]`. Rewiring can only be done if the type arguments are statically known:

```
// before rewiring:
val node1: ListNode[Int] =
      new ListNode[Int](3, null)
// after rewiring:
val node1: ListNode[Int] =
      new ListNode_I(3, null)
// not rewired if U is an abstract type or the
// type parameter of an enclosing class/method
val node2: ListNode[U] =
      new ListNode[U](u, null)
```

The next step of rewiring changes inheritance relations when parent classes have specialized variants that match the type arguments. This injects specialized variants of a class in the inheritance chain, making it possible to use unboxed values when extending a specialized class. This is yet another opportunistic transformation, since the inheritance relation is only rewritten if the type arguments are known statically, as shown by the following example:

```
// before rewiring:
class IntNode(head: Int, tail: IntNode)
      extends ListNode[Int](head, tail)
// after rewiring:
class IntNode(head: Int, tail: IntNode)
      extends ListNode_I(head, tail)
// not rewired, T not known statically:
class MyNode[T](head: T, tail: MyNode[T])
      extends ListNode[T](head, tail)
```

The two rewirings above inject specialized classes in the code. Still, call sites point to the homogeneous methods, which use boxed values. The last rewiring addresses methods, which are rewritten depending on the type of their receiver. Any call site with a specialization-annotated receiver for which the type argument is statically known is rewritten to use specialized versions of the methods. In the first call site of the example below, the receiver is the specialization-

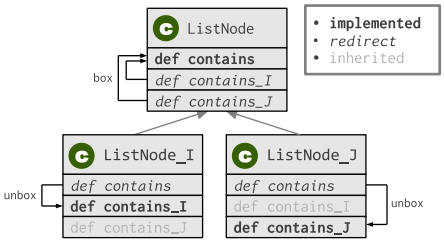


Figure 1.2 – Method overriding and redirection for ListNode and two of its specialized variants. Constructors and accessors are omitted from this diagram.

annotated class `ListNode` and the type argument is statically known to be `Int`. Therefore the call to `contains` is rewired to the specialized `contains_I`:

```
// before rewiring:
(node1: ListNode[Int]).contains(3)
// after rewiring:
(node1: ListNode[Int]).contains_I(3)
// not rewired if U is an abstract type or the
// type parameter of an enclosing class/method
(node2: ListNode[U]).contains(u)
```

1.2.4 Specialization Compatibility

Since the rewiring process only takes place for statically known type arguments, the generic class and its specialized subclasses may be mixed together. In the following snippet, the first branch of the `if` statement is rewired to create an instance of `ListNode_I` while the second branch calls the `node` method, whose type parameter `T` is not annotated for specialization, and thus creates the generic class `ListNode`. Therefore, the value `lst` (of static type `ListNode[Int]`) may be either an instance of `ListNode_I` or of `ListNode`, depending on the random condition:

```
// new ListNode[T] not rewired to
// ListNode_I since T is a type parameter
def node[T](t: T) = new ListNode[T](t, null)

val lst: ListNode[Int] =
  if (Random.nextInt().isEven)
    new ListNode[Int](1, null) // ListNode_I
  else
    node(2)                    // ListNode

lst.contains(0) // rewired to contains_I
```

Therefore, calling a specialized method, `contains_I` in this case, can have as receivers both the generic class, `ListNode`, and the specialized one, `ListNode_I`. So both classes must implement the specialized method. To do so, in `ListNode`, `contains` will be implemented using generic code and `contains_I` will box the argument and call `contains`. In `ListNode_I`, `contains_I` will be implemented using primitive value types and `contains` will unbox and redirect. This can be generalized to multiple specialized variants, as can be seen in Figure 1.2: The generic

class at the top of the hierarchy contains all specialized variants of the `contains` method as redirects to the generic method. Then, each specialized variant of the class inherits from the generic class and overrides its corresponding specialized methods (such as `contains_I` for `ListNode_I`) with the heterogeneously transformed code and redirects the generic method to the specialized variant.

This shows the compatible nature of specialization: in order to avoid boxing, both the call site and the receiver need to be rewired, which means the receiver needs to be specialized and the call site needs to know the type arguments statically or be part of code that will be specialized. But if either condition is not fulfilled, the code remains compatible by boxing, either at the call site itself or inside the redirecting method.

From the perspective of typing the abstract syntax trees, compatibility is achieved because types are assigned before the specialization phase and are not modified later, so they refer to the generic class, even in the presence of rewiring. The first example in §1.2.3 shows that despite rewiring the **new** operator to create an instance of `ListNode_I`, the type of the `node1` value remains `ListNode[Int]`. Thus type-level compatibility is satisfied by `ListNode_I` being a subtype of `ListNode`, and the reverse subtyping is not necessary, as types never refer to `ListNode_I`[3].

1.2.5 Limitations of Specialization

There are two limitations in specialization: the bytecode explosion and the crippled specialized class inheritance. We will describe each problem and show how both can be addressed by the miniboxing encoding.

The specialization mechanism for generating variants is static: whenever the compiler encounters a class annotated for specialization, it generates all its variants up front and outputs bytecode for each of them. This is done to support separate compilation.

Theoretically, the specialized variant creation could be delayed until the actual usage but this requires that the source files for specialized classes are available in all future compilation stages, exactly like in C++. This approach is undesirable from a user perspective, as it also requires encoding the original compilation flags and state, which can influence the generated code. Therefore the simplest, although bytecode-expensive solution was chosen: to generate specialized variants for all value types during compilation.

Fulfilling the bytecode compatibility requirements described before, for n type parameters and full specialization, means the generic class needs to implement 10^n methods, of which $10^n - 2$ are then inherited in the specialized subclasses and 2 are overridden by each of the 10^n subclasses. This makes the bytecode size proportional to 10^n. If the methods were not inherited but defined in each subclass, the bytecode size would be proportional to 10^{2n}.

[3] Except for the **this** type and singleton types in the adapted code.

Still, the generic parent design choice affects inheritance between specialized classes. Figure 1.3 shows an example where the design of specialization bumps into a multiple class inheritance, which is forbidden by Java. In this case, the children inherit from their generic parent, which is suboptimal, since the specialized variants of `MyList` cannot use the specialization in `ListNode`. Experienced Scala programmers might suggest that `MyNode` should be a trait, so it can be mixed in [**?**]. Indeed this solves the multiple inheritance problem, but creates bytecode proportional to 10^{2n}, because the compiler desugars the trait into an interface, and each specialized `MyList_*` class has to implement the methods in that interface. Other more technical problems stem from this design choice too, but could be avoided by having an abstract parent class. For example, fields from the generic class are inherited by the specialized classes, therefore increasing their memory footprint. Constructors also require more complex code because instantiating a specialized class calls the constructor of its parent, the generic class, which needs to be prevented from running, such that side effecting operations in the original class' constructor are not executed twice.

All in all, at the heart of the bytecode explosion problem and thus the other limitations of specialization, lies the large number of variants per type parameter: 10. For two type parameters, full specialization with correct inheritance creates 10^4 times the bytecode. In practice this is not acceptable. Therefore a natural question to ask is how can we reduce the number of variants generated per type parameter? This is the question that inspired miniboxing.

1.3 Miniboxing Encoding

Constraints on the bytecode size currently prevent us from extending the use of specialization in the standard library, namely, to tuples of three elements, to the collections hierarchy and to `Function` traits, which are used in Scala's object oriented representation of functions. Therefore we propose the miniboxing encoding and transformation as a solution to reduce bytecode size and allow library specialization. Along with the encoding, we present a transformation based on the principles of specialization, but using the miniboxed encoding (§1.4) instead of primitive value types.

The miniboxing technique relies on a simple insight: grouping different value types reduces the number of variants necessary in the heterogeneous translation. To this end, we need to group the value types in the language into disjoint sets and for each set designate a value type, also called a storage type, which can encode any type in that set. Notice that this definition is not limited to primitive value types, but can also be used for C-like structs.

Four conditions need to be satisfied for the miniboxing transformation to work:

- All of the value types in the language can be encoded into one or more storage types;

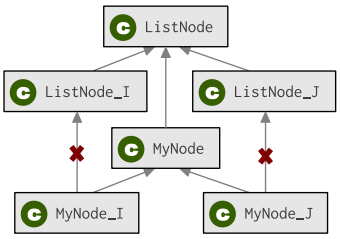


Figure 1.3 – An example of specialized class inheritance made impossible by the current translation scheme.

- The overhead of transforming between any value type and its storage type must be limited, ideally a no-op;
- The operations available for generic types in the language (inherited from the top of the hierarchy, such as `toString`, `hashCode` and `equals`) must be fixed;
- All the value types need to have boxed representations, to enable compatibility between the miniboxed and homogeneous translations (§1.2.4). If the bytecode's common representation is tagged union, the requirement changes to having tagged union representations.

In this case, the heterogeneous translation only needs to generate variants for the storage types and references. References are a special storage type, since all value types are also considered to be part of the reference group. During the translation, whenever a type is not known to be miniboxed to one of the storage types, it is automatically assumed to be attached to the references group. This allows the opportunistic (§1.2.3) and compatible (§1.2.4) rewiring of the tree: indeed since any value type has a boxed representation, it is always correct (but not optimal) to store it as a boxed reference. In the extreme case where all value types are their own storage types, we are back to specialization.

The next subsection will present miniboxing in Scala.

1.3.1 Miniboxing in Scala

In order to apply the miniboxing encoding to Scala, we decided to use the long integer (`Long`) as the storage type of all other primitive value types. Other sets of storage types could also be implemented to improve specific scenarios, such as running on 32-bit architectures (32-bit `Int` and 64-bit `Long`) or using floating-point numerics extensively[4] (64-bit `Double` and 64-bit `Long`). Still, for the rest of the description, we will use the long integer as the only storage type, in order to be consistent with the current implementation of the miniboxing plugin.

[4]The floating point to integer bit-preserving transformations, which are implemented as intrinsics, do incur a measurable overhead.

The transformation primitives from value types to `Long` and back are implemented in the HotSpot Java Virtual Machine and have direct translations to bytecode[4] and to processor instructions [**?**]. Nevertheless, two concerns need our attention when using miniboxing:

- Packing and unpacking cost;
- Memory footprint of the miniboxed encoding.

Packing and unpacking cost. Boxing and unboxing accesses the heap memory. The main goal of miniboxing is to eliminate this overhead, but, in doing so, conversions to and from long integers must not slow down program execution significantly compared to monomorphic code. Our benchmarks show that indeed the overhead is negligible (§1.7).

Memory footprint. The miniboxed encoding has a memory footprint between that of monomorphic and generic code. Considering byte as the type argument, the memory footprint of the miniboxed encoding is 8 times larger than the one for monomorphic code, which would store the byte directly. This factor is reduced by specializing bulk storage (arrays) and considering the paddings introduced by the virtual machine. On the other hand, when compared to boxing on 64 bit processors, the factor is exactly 1, as both a pointer and a long integer have 8 bytes. And this does not take into account the heap space occupied by the boxed values themselves. Therefore, all things considered, miniboxing has a memory footprint larger than the monomorphic and heterogeneous translations, but smaller than homogeneous translations based on boxing.

1.4 Miniboxing Transformation

The miniboxing transformation, which we developed as a Scala compiler plugin, builds upon specialization, which has been formalized in [**?**]. It has the same opportunistic and compatible nature and performs class and method duplication in a similar manner. Still, five elements set it apart:

- the different inheritance scheme (§1.4.1)
- the type bytes for storing encoded types (§1.4.2, §1.4.2)
- the use of a shallow type transformation (§1.4.2)
- the use of the final peephole transformation (§1.4.2)
- the runtime support for miniboxed values (§1.4.3 and §1.5)

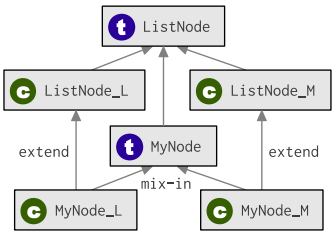


Figure 1.4 – An example of miniboxed class inheritance. The suffixes are: M - miniboxed encoding and L - reference type. Compare to the specialized class inheritance in Figure 1.3.

1.4.1 Inheritance

Miniboxing uses a generic trait as the parent of the specialized classes, therefore avoiding the limitation that miniboxed classes cannot inherit from each other (§1.2.5). Figure 1.4 shows an example miniboxed class inheritance. As explained in §1.2.5, for n specialized type parameters, having a trait as the parent increases the bytecode size from 2^n to 4^n, since each of the 2^n miniboxed variants needs to implement all 2^n methods. Still, the extra bytecode is well spent, for two reasons:

- Having a trait at the top of the hierarchy means no generic fields are inherited in the specialized variants, as it happens when the homogeneous translation is at the top of the hierarchy (§1.2.5);
- This inheritance scheme allows specialized classes to inherit their specialized parent, thus achieving better performance in deep hierarchies.

Since the types assigned to tree nodes do not reference the specialized variants but only the generic interface, this inheritance scheme does not interfere with covariance or contravariance. Indeed, if the type parameter of `ListNode` is defined as covariant, `ListNode_M[Int]` is subtype of `ListNode[Int]` and, transitively, of `ListNode[Any]`.

1.4.2 Miniboxing Specifics

This section will work its way from small examples to describing the new elements in the miniboxing transformation, as compared to specialization. In order to simplify the presentation, we will use the `Long`-based encoding for miniboxing, but the transformation can still be generalized to any number of storage types.

Type Bytes

Type-encoding bytes (or type bytes for short) record the original type of the miniboxed values. Translating the following example shows when type bytes are necessary:

```
def print[@minispec T](value: T): Unit = println(value.toString)
```

Having the type parameter `T` annotated with `@minispec` will trigger miniboxing, which will duplicate this method for `Long`-encoded value types, which we also call miniboxed types. Like specialization, miniboxing produces groups of overloaded methods, with the original method being the all-reference implementation in its group. In our case, only the miniboxed overload needs to be created. To do so, the compiler will create another version of `print` for long integers, which we call `print_M`:

```
def print_M(value: Long): Unit = println(value.toString)
```

This is a very naive translation. Calling `print(false)`, after method rewiring, will transform the boolean to a long integer whose value will be printed on the screen instead of the "false" string. To perform the correct action, the translation should recover the string representation of the boolean value `false` from the `Long` encoding. This suggests the `toString` operation should be rewritten to:

```
def print_M(value: Long): Unit = println(MBRuntime.toString(value))
```

The code above shows a less naive implementation, since it rewires `toString` calls on the miniboxed value to a special runtime support object in order to obtain the string representation. But passing a single miniboxed value isn't enough, as we mentioned miniboxing does not encode the type with the value as tagged unions do [**?**]. Therefore, it should have a separate parameter to encode the original type:

```
def print_M(T_Type: Byte, value: Long): Unit = println(MBRuntime.toString(value,
T_Type))
```

This is close to the minibox-transformed version of `print_M` the plugin would output. The `T_Type` field only encodes the 9 primitive types in Scala, therefore it does not incur the typical overhead of full reified generics [**?**]. A call to `print(false)` will be translated to the following code, where `BOOLEAN` is the type byte for boolean values:

```
print_M(BOOLEAN, MBRuntime.BoolToMinibox(false))
```

The method call above shows two differences between rewiring in miniboxing and specialization:

1. Calling a miniboxing-transformed method (or instantiating a miniboxing-transformed class) requires passing type bytes for all the `Long`-encoded type arguments;

2. The arguments to minibox-transformed methods need to be explicitly encoded in the storage type.

We will now present exactly how the miniboxing plugin arrives to this transformed code. As the miniboxing transformation takes place, it needs to preserve program semantics and type correctness. In order to do so, the transformation for `print` is actually done in three steps.

First, the new signature is created, knowing the type parameter `T` is encoded as `Long`. The method name is mangled (mangled names are simplified in this presentation) and the type byte for `T` is added to the signature. Then parameters are added, with all parameters of type `T` being replaced by parameters of type `Long`. As this happens, the symbols whose types changed are recorded and treated specially. In this case, the only miniboxed parameter is `value`, which is recorded. It is also recorded that the type byte `T_Type` corresponds to the encoded type `T`. This yields: (we'll see later why the type parameter `T` still appears)

```
1 def print_M[T](T_Type: Byte, value: Long): Unit = // need to copy and adapt body
  from print
```

In the second step, the body is copied from the `print` method. To maintain type correctness, all the symbols previously recorded as having their types changed are now automatically boxed back to generic type `T`, so the newly generated code tree is consistent in terms of types:

```
1 def print_M[T](T_Type: Byte, value: Long): Unit =
  println(MBRuntime.MiniboxToBox[T](value, T_Type).toString)
```

In the final step, the tree rewrite rules will transform the call to `MiniboxToBox` followed by `toString` into a single call to the `MBRuntime` system, which typically yields better performance:

```
1 def print_M[T](T_Type: Byte, value: Long): Unit = println(MBRuntime.toString(value,
  T_Type))
```

The next section will explain why it is necessary to carry the type parameter `T`.

Shallow and Deep Type Transformations

To further understand the miniboxing transformation, let us look at a more complex example, which builds a linked list with a single element:

```
1 def list[@minispec T](value: T): ListNode[T] = new ListNode[T](value, null)
```

As explained before, the `list` method will become the all-reference overload. But the interesting transformation happens in the miniboxed variant. If specialization were to transform this method its signature would be:

```
1 def list_M[T](value: Long): ListNode[Long]
```

The return type is incorrect, as we expect `list(3)` to return a `ListNode[Int]`, and yet rewiring `list(3)` to `list_M(...)` would return a `ListNode[Long]`. This exposes the difference between the deep type transformation in specialization and the shallow type transformation in

miniboxing: In miniboxing, only values of type `T` are transformed to `Long`, but any type referring to `T`, such as `ListNode[T]`, will remain the same. This explains why the type parameter `T` is carried over to `print_M` and `list_M`: it may still be used in the method's signature and code. The full transformation for method `list_M` will be:

```
def list_M[T](T_Type: Byte, value: Long): ListNode[T] =
  new ListNode[T](MiniboxToBox[T](value, T_Type))
```

The shallow type transformation also changes types of local variables from `T` to `Long` and recursively transforms all nested methods and classes within the piece of code it is adapting. This propagates the storage type representation throughout the code.

Peephole Transformation

The last transformation to touch the code before it is shipped to the next phase is the peephole transformation, which performs a final sweep over the code to remove redundant conversions. To show this phase at work, let us consider what happens if the `ListNode` class in the last example is also annotated for miniboxing. In this case, the class will have a miniboxed variant, `ListNode_M` to which the instantiation is rewired. Since the `head` parameter of the `ListNode` constructor is boxed, while the `head` parameter of the `ListNode_M` constructor is miniboxed, the transformation will introduce a new `BoxToMinibox` conversion:

```
def list_M[T](T_Type: Byte, value: Long): ListNode[T] =
  new ListNode_M[T](T_Type, BoxToMinibox[T](MiniboxToBox[T](value, ...)), null)
```

Converting from `Long` to the boxed representation and back before creating the list node will certainly affect performance. Such consecutive complementary conversions and other suboptimal constructs are automatically removed by the peephole optimization:

```
def list_M[T](T_Type: Byte, value: Long): ListNode[T] =
  new ListNode_M[T](T_Type, value, null)
```

The code produced by the rewiring phase can be optimized by a single pass of the peephole transformation so there is no need to iterate until a fixed point is reached.

Type Bytes in Classes

The class translation is slightly more complex than method translation. For classes, type bytes are also included as fields in the miniboxed variants, to allow the class' methods to encode and decode miniboxed values as necessary:

```
class ListNode[@minispec T]
  (val head: T, val tail: ListNode[T]) {
  def contains(element: T): Boolean = ...
}
```

The interface resulting after miniboxing will be:

```
trait ListNode[T] {
  ... // getters for head and tail
  def contains(element: T): Boolean
  def contains_M(T_Type_local: Byte, element: Long): Boolean
}
```

And the miniboxed variant of this class will be:

```
class ListNode_M[T]
  (T_Type: Byte, head: Long, tail: ListNode[T]) extends ListNode[T] {
  ... // getters for head and tail
  def contains(element: T): Boolean =
       ... // redirect to this.contains_M
  def contains_M(T_Type_local: Byte, element: Long): Boolean =
       ... // specialized implementation
}
```

`ListNode_M` has two type tags: `T_Type` is a class parameter and becomes a field of the class while `T_Type_local` is passed to the `contains_M` method directly. In the code example, `T_Type` is used to convert the `element` parameter of `contains` to its miniboxed representation when redirecting the call to `contains_M`. But `T_Type_local` is not used in the `ListNode_M` class. To understand when `T_Type_local` is necessary, we have to look at the reference-carrying variant of the `ListNode` class:

```
class ListNode_L[T]
  (head: T, tail: ListNode[T]) extends ListNode[T] {
  ... // getters for head and tail
  def contains(element: T): Boolean =
       ... // generic implementation
  def contains_M(T_Type_local: Byte, element: Long): Boolean =
       ... // redirect to this.contains
}
```

All instantiations of `ListNode` where the type argument is statically known to be a value type are rewired to `ListNode_M`. The rest of the instantiations are rewired to `ListNode_L`, either because the type argument is not known statically or because it is known to be a reference type. Therefore, there is no reason for `ListNode_L` to carry `T_Type` as a global field. But, in order to allow `contains_M` to decode the miniboxed value `element` into a boxed form and redirect the call `contains`, a local type byte is necessary. Since the `ListNode` interface and its two implementations, `ListNode_L` and `ListNode_M` need to be compatible, the local type byte in `contains_M` is also present for `ListNode_M`, although in the miniboxed class it is redundant.

1.4.3 Calling the Runtime Support

The previous examples have shown how the miniboxing plugin uses the `MBRuntime` object for conversions between unboxed, miniboxed and boxed data representations. But the `MBRuntime` object is not limited to conversions. In Scala, any type parameter is assumed to be a subtype

of the `Any` class, so the programmer can invoke methods such as `toString`, `hashCode` and `equals` on generic values. As shown in §1.4.2, these calls can be translated by a conversion to the boxed representation followed by the call, but are further optimized by calling the implementations in `MBRuntime`, which work directly on miniboxed values.

Aside from conversions and implementations for the methods in the `Any` class, the miniboxing runtime support contains code to allow direct interaction between arrays and miniboxed values. An example that uses arrays is the `ArrayBuffer` class:

```
class ArrayBuffer[@minispec T: Manifest] {
  // fields:
  private[this] var array = new Array[T](32)
  ...
  // methods:
  def getElement(idx: Int): T = array(idx)
  ...
}
```

The miniboxed variant `ArrayBuffer_M` is rewritten to call the `MBArray` object to create and access arrays in the miniboxed format:

```
// ArrayBuffer miniboxed variant for primitives:
class ArrayBuffer_M[T: Manifest](T_Type: Byte)
                      extends ArrayBuffer[T] {
  // fields:
  private[this] var array: Array[T] = MBArray.mbarray_new(32, T_Type)
  ...
  // methods:
  def getElement(idx: Int): T =
     MiniboxToBox(getElement_M(T_Type, idx), ...)
  def getElement_M(T_Type_local: Byte, idx: Int): Long =
     MBArray.array_get(array, idx, T_Type)
  ...
}
```

The implementation of the `MBArray` object is critical to numeric algorithms and performance data structures, as it has to be small enough to be inlined by the just-in-time compiler and structured in ways that return the result as fast as possible for any of the primitive types. The following two sections describe the runtime support for arrays and give technical insights into the pitfalls of the implementation.

1.5 Miniboxing Bulk Storage Optimization

Arrays are Java's bulk storage facility. They can store value types or references to heap objects. This is done efficiently, as values are stored one after the other in contiguous blocks of memory and access is done in constant time. Their characteristics make arrays good candidates for internal data structures in collections and algorithms.

But in order to implement compact storage and constant access overhead, arrays are monomorphic under the hood, with separate (and incompatible) variants for each of the primitive value types. What's more, each array type has its own specific bytecode instructions to manipulate it.

The goal we set forth was to match the performance of monomorphic arrays in the context of miniboxing-encoded values. To this end, we had two alternatives to implementing arrays for miniboxed values: use arrays of long integers to store the encoded values or use monomorphic arrays for each type, and encode or decode values at each access.

Storing encoded values in arrays provides the advantage of uniformity: all the code in the minibox-specialized class uses the `Long` representation and array access is done in a single instruction. Although this representation wastes heap space, especially for small value types such as boolean or byte, this is not the main drawback: it is incompatible with the rest of the Scala code.

In order to stay compatible with Java, Scala code uses monomorphic arrays for each value type. Therefore arrays of long integers in miniboxed classes must not be allowed to escape from the transformed class, otherwise they may crash outside code attempting to read or write them. To maintain compatibility, we could convert escaping arrays to their monomorphic forms. But the conversion would introduce delays and would break aliasing, as writes from the outside code would not be visible in the miniboxed code and vice versa. Since completely prohibiting escaping arrays severely restricts the programs that can use miniboxing, this solution becomes unusable in practice.

Thus, the only choice left is to use arrays in their monomorphic format for each value type, so we maintain compatibility with the rest of the Scala code. This decision led to another problem: any array access requires a call to the miniboxing runtime support which performs a dispatch on the type byte. Depending on the type byte's value, the array is cast to its correct type and the corresponding bytecode instruction for accessing it is used. This is followed by the encoding operation, which converts the read value to a long integer. The following snippet shows the array read operation implemented in the miniboxing runtime support code:

```
1 def array_get[T](array: Array[T], idx: Int, tag: Byte): Minibox = tag match {
2   case INT =>
3     array.asInstanceOf[Array[Int]](idx).toLong
4   case LONG =>
5     array.asInstanceOf[Array[Long]](idx)
6   case DOUBLE => Double.doubleToRawLongBits(
7     array.asInstanceOf[Array[Double]](idx)).toLong
8   ...
9 }
```

The most complicated and time-consuming part of our work involved rewriting the miniboxing runtime support to match the performance of specialized code. The next subsections present the HotSpot Java Virtual Machine execution (§1.5.1), the main benchmark we used for testing

(§1.5.2) and two implementations for the runtime support: type byte switching (§1.5.3) and object-oriented dispatching (§1.5.4).

1.5.1 HotSpot Execution

We used benchmarks to guide our implementation of the miniboxing runtime support. In this section we will briefly present the just-in-time compilation and optimization mechanisms in the HotSpot Java Virtual Machine [**?**, **?**], since they directly influenced our design decisions. Although the work is based on the HotSpot Java Virtual Machine, we highlight the underlying mechanisms that interfere with miniboxing, in hope that our work can be used as the starting point for the analysis on different virtual machines.

The HotSpot Java Virtual Machine starts off by interpreting bytecode. After several executions, a method is considered "hot" and the just-in-time compiler is called in to transform it into native code. During compilation, aggressive inlining is done recursively on all the methods that have been both executed enough times and are small enough. Typical inlining requirements for the C2[5] (server) just-in-time compiler are 10000 executions and size below 35 bytes.

When inlining static calls, the code is inlined directly. For virtual and interface calls, however, the code depends on the receiver. To learn which code to inline, the virtual machine will profile receiver types during the interpretation phase. Then, if a single receiver is seen at runtime, the compiler will inline the method body from that receiver. This inlining may later become incorrect, if a different class is used as the receiver. For such a case the compiler inserts a guard: if the runtime type is not the one expected, it jumps back to interpretation mode. The bytecode may be compiled again later if it runs enough times, with both possible method bodies inlined. But if a third runtime receiver is seen, the call site is marked as megamorphic and inlining is not performed anymore, not even for the previous two method bodies.

After inlining as much code as feasible, the virtual machine's just-in-time compiler applies optimizations, which significantly reduce the running time, especially for array operations which are very regular and for which bounds checks can be eliminated.

1.5.2 Benchmark

We benchmarked the performance on the two examples previously shown in the paper, `ListNode` and `ArrayBuffer`. Throughout benchmarking, one particular method stood out as the most sensitive to the runtime support implementation: the `reverse` method of the `ArrayBuffer` class. The rest of this section uses the `reverse` method to explore the performance of different implementations of the runtime support:

[5] We did not use tiered compilation.

	Single Context	Multi Context
generic	20.4	21.5
miniboxed, no inlining	34.5	34.4
miniboxed, full switch	2.4	15.1
miniboxed, semi-switch	2.4	17.2
miniboxed, decision tree	24.2	24.1
miniboxed, linear	24.3	22.9
miniboxed, dispatcher	2.1	26.4
specialized	2.0	2.4
monomorphic	2.1	N/A

Table 1.1 – The time in milliseconds necessary for reversing an array buffer of 3 million integers. Performance varies based on how many value types have been used before (Single Context vs. Multi Context).

```
1  def reverse_M(T_Type_local: Byte): Unit = {
2    var idx = 0
3    val xdi = elemCount - 1
4    while (idx < xdi) {
5      val el1: Long = getElement_M(T_Type, idx)
6      val el2: Long = getElement_M(T_Type, xdi)
7      setElement_M(T_Type, idx, el2)
8      setElement_M(T_Type, xdi, el1)
9      idx += 1
10     xdi -= 1
11   }
12 }
```

The running times presented in table 1.1 correspond to reversing an integer array buffer of 3 million elements. To put things into perspective, along with different designs, the table also provides running times for monomorphic code (specialized by hand), specialization-annotated code and generic code. Measurements are taken in two scenarios: For “Single Context”, an array buffer of integers is created and populated and its `reverse` method is benchmarked. In “Multi Context”, the array buffer is instantiated, populated and reversed for all primitive value types first. Then, a new array buffer of integers is created, populated and its `reverse` method is benchmarked. The HotSpot Java Virtual Machine optimizations are influenced by the historical paths executed in the program, so using other type arguments can have a drastic impact on performance, as can be seen from the table, where the times for “Single Context” and “Multi Context” are very different: this means the virtual machine gives up some of its optimizations after seeing multiple instantiations with different type arguments. “Multi Context” is the likely scenario a library class will be in, as multiple instantiations with different type arguments may be created during execution.

1.5.3 Type Byte Switching

The first approach we tried, the simple switch on the type byte, quickly revealed a problem: The array runtime support methods were too large for the just in time compiler to inline

at runtime. This corresponds to the "miniboxing, no inlining" in table 1.1. To solve this problem, we tasked the Scala compiler with inlining runtime support methods in its backend, independently of the virtual machine. But this was not enough: the `reverse_M` method calls `getElement_M` and `setElement_M`, which also became large after inlining the runtime support, and were not inlined by the virtual machine. This required us to recursively mark methods for inlining between the runtime support and the final benchmarked method.

The forced inlining in the Scala backend produced good results. The measurement, corresponding to the "miniboxed, full switch" row in the table, shows miniboxed code working at almost the same speed as specialized and monomorphic code. This can be explained by the loop unswitching optimization in the just-in-time compiler. With all the code inlined by the Scala backend, loop unswitching was able to hoist the type byte switch statement outside the while loop. It then duplicated the loop contents for each case in the switch, allowing array-specific optimizations to bring the running time close to monomorphic code.

But using more primitive types as type arguments diminished the benefit. We tested the `reverse` operation in two situations, to check if the optimizations still take place after we use it on array buffers with different type arguments. It is frequently the case that the HotSpot Java Virtual Machine will compile a method with aggressive assumptions about which paths the execution may take. For the branches that are not taken, guards are left in place. Then, if a guard is violated during execution, the native code is interrupted and the program continues in the interpreter. The method may be compiled again later, if it is executed enough times to warrant compilation to native code. Still, upon recompilation, the path that was initially compiled to a stub now becomes a legitimate path and may preclude some optimizations. We traced this problem to the floating point encoding, specifically the bit-exact conversion from floating point numbers to integers, that, once executed, prevents loop unswitching.

We tried different constructions for the miniboxing runtime support: splitting the match into two parts and having an if expression that would select one or the other ("semi-switch"), transforming the switch into a decision tree ("decision tree") and using a linear set of 9 if statements ("linear"), all of which appear in table 1.1. These new designs either degraded in the multiple context scenario, or provided a bad baseline performance from the beginning. What's more, the fact that the runtime "remembered" the type arguments a class was historically instantiated with made the translation unusable in practice, since this history is not only influenced by code explicitly called before the benchmark, but transitively by all code executed since the virtual machine started.

1.5.4 Dispatching

The results obtained with type byte switching showed that we were committing to a type too late in the execution: Forced inlining had to carry our large methods that covered all types inside the benchmarked method, where the optimizer had to hoist the switch outside the loop:

```
while (...) {
  val el1: Long = T_Type match { ... }
  val el2: Long = T_Type match { ... }
  T_Type match { ... }
  T_Type match { ... }
}
```

Ideally, this switch should be done as early as possible, even as soon as class instantiation. This can be done using an object-oriented approach: instead of passing a byte value during class instantiation and later switching on it, we can pass objects which encode the runtime operations for a single type, much like the where objects in PolyJ [**?**]. We call this object the dispatcher. The dispatcher for each value type encodes a common set of operations such as array get and set. For example, `IntDispatcher` encodes the operations for integers:

```
abstract class Dispatcher {
  def array_get[T](arr: Array[T], idx: Int): Long
  def array_update[T](arr: Array[T], idx: Int, elt: Long): Unit
  ...
}
object IntDispatcher extends Dispatcher { ... }
```

Dispatcher objects are passed to the miniboxed class during instantiation and have final semantics. In the `reverse` benchmark, this would replace the type byte switches by method invocations, which could be inlined. Dispatchers make forced inlining and loop unswitching redundant. With the final `dispatcher` field set at construction time, the `reverse_M` inner loop body can have array access inlined and optimized: ("miniboxed, dispatcher" in tables 1.1 and 1.2)

```
// inlined getElement:
val el1: Long = dispatcher.array_get(...)
val el2: Long = dispatcher.array_get(...)
// inlined setElement:
dispatcher.array_update(...)
dispatcher.array_update(...)
```

Despite their clear advantages, in practice dispatchers can be used with at most two different value types. This happens because the HotSpot Java Virtual Machine inlines the dispatcher code at the call site and installs guards that check the object's runtime type. The inline cache works for two receivers, but if we try to swap the dispatcher a third time, the callsite becomes megamorphic. In the megamorphic state, the `array_get` and `array_set` code is not inlined, hence the disappointing results for the "Multi Context" scenario.

Interestingly, specialization performs equally well in both "Single Context" and "Multi Context" scenarios. The explanation lies in the bytecode duplication: each specialized class contains a different body for the reverse method, and the profiles for each method do not interact. Accordingly, the results for integers are not influenced by the other value types used. This insight motivated the load-time cloning and specialization, which is described in the next section.

	Single Context	Multi Context
generic	20.4	21.5
miniboxed, full switch	2.4	15.1
mb. full switch, LS	2.5	**2.4**
miniboxed, dispatcher	2.1	26.4
mb. dispatcher, LS	2.0	**2.7**
specialized	2.0	2.4
monomorphic	2.1	N/A

Table 1.2 – The time in milliseconds necessary for reversing an array buffer of 3 million integers. Miniboxing benchmarks ran with the double factory mechanism and the load-time specialization are marked with LS.

1.6 Miniboxing Load-time Optimization

The miniboxing runtime support, in both incarnations, using switching and dispatching, fails to deliver performance in the "Multi Context" scenario. The reason, in both cases, is that execution takes multiple paths through the code and this prevents the Java Virtual Machine from optimizing. Therefore an obvious solution is to duplicate the class bytecode, but instead of duplicating it on the disk, as specialization does, we do it in memory, on-demand and at load-time. The .NET Common Language Runtime [**?**, **?**] performs on-demand specialization at load-time, but it does so using more complex transformations encoded in the virtual machine. Instead, we use Java's classloading mechanism.

We use a custom classloader to clone and specialize miniboxed classes. Similar to the approach in *Pizza* [**?**], the classloader takes the name of a class that embeds the type byte value. For example, `ListNode_I` corresponds to a clone of `ListNode_M` with the type byte set to `INT`. From the name, the classloader infers the miniboxed class name and loads it from the classpath. It clones its bytecode and adjusts the constant table [**?**]. All this is done in-memory.

Once the bytecode is cloned, the paths taken through the inlined runtime support in each class remain fixed during its lifetime, making the performance in "Single Context" and "Multi Context" comparable, as can be seen in Table 1.2. The explanation is that the JVM sees different classes, with separate type profiles, for each primitive type.

Aside from bytecode cloning, the classloader also performs class specialization:

- Replaces the type tag fields by static fields (as the class is already dedicated to a type);
- Uses constant propagation and dead code elimination to reduce each type tag switch down to a single case, which can be inlined by the virtual machine, thus eliminating the need for forced inlining;
- Performs load-time rewiring, which is described in the next section.

1.6.1 Miniboxing Load-time Rewiring

When rewiring, the miniboxing transformation follows the same rules set forth by specialization (§1.2.3). Load-time cloning introduces a new layer of rewiring, which needs to take the cloned classes into account. The factory mechanism we employ to instantiate cloned and specialized classes (§1.6.2) is equivalent to the instance rewiring in specialization. The two other rewiring steps in specialization are method rewiring and parent class rewiring. Fortunately method rewiring is done during compilation and since methods are not modified, there is no need to rewire them in the classloader. Parent classes, however, must be rewired at load-time to avoid performance degradation.

Load-time parent rewiring allows classes to inherit and use miniboxed methods while keeping type profiles clean. If the parent rewiring is done only at compile-time, all classes extending `ArrayBuffer_M` share the same code for the `reverse_M` method. But since they may use different type arguments when extending `ArrayBuffer`, they are back to the "Multi Context" scenario in table 1.1. To obtain good performance, rewiring parent classes is done first at compile time, to the miniboxed variant of the class, and then at load-time, to the cloned and specialized class. The following snippet shows parent rewiring in the case of dispatcher objects:

```
// user code:
class IntBuff extends ArrayBuffer[Int]
// after compile-time rewiring:
class IntBuff extends ArrayBuffer_M[Int](IntDispatcher)
// after load-time rewiring:
class IntBuff extends ArrayBuffer_I[Int](IntDispatcher)
```

The load-time rewiring of parent classes requires all subclasses with miniboxed parents to go through the classloader transformation. This includes the classes extending miniboxed parents with static type arguments, such as the `IntBuff` class in the code snippet before. This incurs a first-instantiation overhead, which is an inconvenience especially for classes that are only used once, such as anonymous closures extending `FunctionX`. But not all classes make use of the miniboxing runtime for arrays, so we can devise an annotation which hints to the compiler which classes need factory instantiation. This would only incur the cloning and specialization overhead when the classes use arrays. The annotation could be automatically added by the compiler when a class uses array operations and propagated from parent classes to their children:

```
@loadtimeSpec
class ArrayBuffer[@minispec T]

// IntBuff automatically inherits @loadtimeSpec
class IntBuff extends ArrayBuffer[Int]
```

1.6.2 Efficient Instantiation

Imposing the use of a global classloader is impossible in many practical applications. To allow miniboxing to work in such cases, we chose to perform the class instantiation through a factory that loads a local specializing classloader, requests the cloning and specialization of the miniboxed class and instantiates it via reflection. We benchmarked the approach and it introduced significant overhead, as instantiations using reflection are very expensive.

To counter the cost of reflective instantiation, we propose a "double factory" approach that uses a single reflective instantiation per cloned class. In this approach each cloned and specialized class has a corresponding factory – that instantiates it using the **new** keyword. When instantiating a miniboxed class with a new set of type arguments, its corresponding factory is specialized by the classloader and instantiated via reflection. From that point on, any new instance is created by the factory, without the reflective delay. The following code snippet shows the specialized (or 2nd level) factory:

```
1  // Factory interface
2  abstract class ArrayBufferFactoryInterface {
3    def newArrayBuffer_M[T: Manifest](disp: Dispatcher[T]): ArrayBuffer[T]
4  }
5  // Factory instance, to be specialized
6  // in the classloader
7  class ArrayBufferFactoryInstance_M extends ArrayBufferFactoryInterface {
8    def newArrayBuffer_M[T: Manifest](disp: Dispatcher[T]): ArrayBuffer[T] =
9      new ArrayBuffer_M(disp)
10 }
```

1.7 Evaluation

This section presents the results obtained by the miniboxing transformation. It will first present the miniboxing compiler plug-in and the miniboxing classloader (§1.7.1). Next, it will present the benchmarking infrastructure (§1.7.2) and the benchmark targets (§1.7.3). Finally, it will present the results (§1.7.4 - §1.7.8) and draw conclusions (§1.7.9).

1.7.1 Implementation

The miniboxing plug-in adds a code transformation phase in the Scala compiler. Like specialization, the miniboxing phase is composed of two steps: transforming signatures and transforming trees. As the signatures are specialized, metadata is stored on exactly how the trees need to be transformed. This metadata later guides the tree transformation in duplicating and adapting the trees to obtain the miniboxed code. The duplication step reuses the infrastructure from specialization, with a second adaptation step which transforms storage from generic to miniboxed representation.

The plugin performs several transformations:

- Code duplication and adaptation, where values of type `T` are replaced by long integers and are un-miniboxed back to `T` at use sites (§1.4.2);
- Rewiring methods like `toString`, `hashCode`, `equals` and array operations to use the runtime support (§1.4.3);
- Opportunistic rewiring: new instance creation, specialized parent classes and method invocations (§1.2.3);
- Peephole minibox/un-minibox reduction (§1.4.2).

The miniboxing classloader duplicates classes and performs the specialized class rewiring. It uses transformations from an experimental Scala backend to perform constant propagation and dead code elimination in order to remove switches on the type byte. It supports miniboxed classes generated by the current plug-in and in the current release only works for a single specialized type parameter. Also, the infrastructure for the double factory instantiation was written and tuned by hand, and may be integrated in the plug-in in a future release. We did not implement the `@loadtimeSpec` annotation yet.

The project also contains code for testing the plug-in and the classloader and performing microbenchmarks, something which turned out to be more difficult than expected.

1.7.2 Benchmarking Infrastructure

The miniboxing plug-in produces bytecode which is then executed by the HotSpot Java Virtual Machine. Although the virtual machine provides useful services to the running program, such as compilation, deoptimization and garbage collection, these operations influence our microbenchmarks by delaying or even changing the benchmarked code altogether. Furthermore, the non-deterministic nature of such events make proper benchmarking harder [**?**].

In order to have reliable results for our microbenchmarks, we used ScalaMeter [**?**], a tool specifically designed to reduce benchmarking noise. ScalaMeter is currently used in performance-testing the Scala standard library. When benchmarking, it forks a new virtual machine such that fresh code caches and type profiles are created. It then warms up the benchmarked code until the virtual machine compiles it down to native code using the C2 (server) [**?**] compiler. When the code has been compiled and the benchmark reaches a steady state, ScalaMeter measures several execution runs. The process is repeated several times, 100 in our case, reducing the benchmark noise. For the report, we present the average of the measurements performed.

We ran the benchmarks on an 8-core i7 machine running at 3.40GHz with 16GB of RAM memory. The machine ran a 64 bit version of Linux Ubuntu 12.04.2. For the Java Virtual Machine we used the Oracle Java SE Runtime Environment build 1.7.0_11 using the C2 (server) compiler. The following section will describe the benchmarks we ran.

1.7.3 Benchmark Targets

We executed the benchmarks in two scenarios:

- "Single Context" corresponds to the benchmark target (`ArrayBuffer` or `ListNode`) executed with a single value type, `Int`;
- "Multi Context" corresponds to running the benchmark for all value types and only then measuring the execution time for the target value type, `Int`;

The benchmarks were executed with 7 transformations:

- generic: the generic version of the code, uses boxing;
- mb. switch: miniboxed, using the type byte switching;
- mb. dispatcher: miniboxed, dispatcher runtime support;
- mb. switch + LS: miniboxed, type byte switching, load-time specialization with the double factory mechanism;
- mb. dispatcher + LS: miniboxed, dispatcher, load-time specialization with the double factory mechanism;
- specialized: code transformed by specialization;
- monomorphic: code specialized by hand, which does not need the redirects generated by specialization.

For the benchmarks, we used the two classes presented in the previous sections: The `ArrayBuffer` class simulates collections and algorithms which make heavy use of bulk storage and the `ListNode` class simulates collections which require random heap access. We chose the benchmark methods such that each tested a certain feature of the miniboxing transformation. We used very small methods such that any slowdowns can easily be attributed to bytecode or can be diagnosed in a debug build of the virtual machine, using the compilation and deoptimization outputs.

`ArrayBuffer.append` creates a new array buffer and appends 3 million elements to it. This benchmark tests the array writing operations in isolation, such that they cannot be grouped together and optimized.

`ArrayBuffer.reverse` reverses a 3 million element array buffer. This benchmark proved the most difficult in terms of matching the monomorphic code performance.

`ArrayBuffer.contains` checks for the existence of elements inside an initialized array buffer. It exercises the `equals` method rewiring and revealed to us that the initial transformation for

`equals` was suboptimal, as we were not using the information that two miniboxed values were of the same type. This benchmark showed a 22x speedup over generic code.

List construction builds a 3 million element linked list using `ListNode` instances. This benchmark verifies the speed of miniboxed class instantiation. It was heavily slowed down by the reflective instantiation, therefore we introduced the double factory for class instantiation using the classloader.

`List.hashCode` computes the hash code of a list of 3 million elements. We used this benchmark to check the performance of the `hashCode` rewiring. It was a surprise to see the `hashCode` performance for generic code running in the interpreter (Table 1.4). It is almost one order of magnitude faster than specialized code and 5 times faster than miniboxing. The explanation is that computing the hash code requires boxing and calling the `hashCode` method on the boxed object. When the benchmarks are compiled and optimized, this is avoided by inlining and escape analysis, but in the interpreter, the actual object allocation and call to `hashCode` do happen, making the heterogeneous translation slower.

`List.contains` tests whether a list contains an element, repeated for 3 million elements. It tests random heap access and the performance of the `equals` operator rewiring.

1.7.4 Benchmark Results

Table 1.3 presents the main results of our benchmarks. The table highlights “mb. switch + LS” and “mb. dispatch + LS”, which represent the miniboxing encoding using the load-time specialization invoked with the double factory mechanism.

The miniboxing encoding based on type tag switching, “mb. switch + LS”, offers steady performance close to that of specialization and monomorphic code, with slowdowns ranging between 0 and 20 percent. The classloader specialization, coupled with constant propagation and dead code elimination, make the type tag switching approach the most stable across multiple executions with different type arguments, with at most 6 percent difference between “Single Context” and “Multi Context”, in the case of `ArrayBuffer.append`.

The dispatcher-based encoding, “mb. dispatch + LS”, also offers performance close to specialization and monomorphic code, with slightly better performance when traversing the linked list (benchmarks `hashCode` and `contains`), and a lower performance on `List` creation. This suggests that passing the dispatcher object on the stack is more expensive than passing a type tag.

It is worth noting that the dispatcher-based implementation relies on inlining performed by the just-in-time compiler. Although the load-time cloning mechanism ensures type profiles remain monomorphic, the burden of inlining falls on the just-in-time compiler. In the case of virtual machines that perform ahead-of-time compilation, such as Excelsior JET [**?**], the newly specialized class is compiled to native code without interpretation, thus no type profiles are

available and no inlining takes place for the miniboxing runtime. In contrast to dispatching, type tag switching only requires loading-time constant propagation and dead code elimination to remove the overhead of the miniboxing runtime. This makes it a better candidate for robust performance across different virtual machines. The next section will present interpreter benchmarks.

1.7.5 Interpreter Benchmarks

Before compiling the bytecode to native machine code, the HotSpot Virtual Machine interprets it and gathers profiles that later guide compilation. Table 1.4 presents results for running the same set of benchmarks in the interpreter, without compilation. It is important that transformations do not visibly degrade performance in the interpreter, as this slows down application startup. The data highlights a steady behavior for the the type tag switching, while the dispatcher-based approach suffers from up to 4x slowdowns.

The data shows a consistent slowdown of the tag switching approach compared to the monomorphic code in 4 of the 6 experiments. This can most likely be attributed to the mechanism for invoking object methods, which requires loading a reference to the module from a static field and then performing a method call. Even after the method call is inlined, the Scala backend (and the load-time specializer) do not remove the static field access, thus leaving the redundant but possibly side-effecting instruction in the hot loop. In the native code the field access is compiled away by the just-in-time compiler. This could be improved in the Scala backend.

1.7.6 Bytecode Size

Table 1.5 presents the bytecode generated for `ArrayBuffer` and `ListNode` by 4 transformations: erasure, miniboxing with dispatcher, miniboxing with switching and specialization. The fraction of bytecode created by miniboxing, when compared to specialization, lies between 0.2x to 0.4x. This is marginally better than the fraction we expected, 0.4x, which corresponds to $4^n/10^n$ for $n = 1$. The reason the fraction is $4^n/10^n$ instead of $2^n/10^n$ is explained in §1.4.1. The double factory mechanism adds a significant bytecode, in the order of 10 kilobytes per class.

In order to evaluate the benefits of using the miniboxing encoding for real-world software, we developed a "specialization-hijacking" mode, where specialization was turned off and all `@specialized` notations were treated as `@minispec`, thus triggering miniboxing on all methods and classes where specialization was used. For this benchmark we only used the switching-based transformation.

The first evaluation was performed on Spire [**?**], a Scala library providing abstractions for numeric types, ranging from boolean algebras to complex number algorithms. Spire is the one library in the Scala community which uses specialization the most, and the project owner,

Erik Osheim, contributed numerous bug fixes and enhancements to the Scala compiler in the area of specialization. The results, presented in Table 1.6, show a bytecode reduction of 2.8x and a 1.4x, or 40%, reduction in the number of specialized classes. The two reductions are not proportional because specialized methods inflate the code size of classes, but do not increase the class count. The bytecode reduction is limited to 2.8x because specialization is used in a directed manner, pointing exactly to the value types which should be specialized. So, instead of generating 10 classes per type parameter, it only generates the necessary value types. Nevertheless, even starting from manually directed specialization, the miniboxing transformation is able to further reduce the bytecode size.

The second evaluation, shown in Table 1.7, is motivated by a common complaint in the Scala community: that the collections in the standard library should be specialized. To perform an evaluation on collections, we sliced a part of the library around the `Vector` class and examined the impact of using the specialization and miniboxing transformations. On the approximately 64 Scala classes, traits and objects included in our slice, the bytecode reduction obtained by miniboxing compared to specialization is 4.7x. Compared to the generic `Vector`, the miniboxing code growth is 1.7x, opposed to almost 8x for specialization.

1.7.7 Load-time Specialization Overhead

In this section we will evaluate the overhead of the double factory mechanism. There are three types of overhead involved:

- Bytecode overhead, shown in the previous section;
- Time spent specializing and loading a class;
- Heap overhead for the classloader and factory.

We will further explore the last two sources of overhead.

Time Spent Specializing

Table 1.3, in the "`List` creation" column, shows the overhead of the double factory mechanism and class specialization is not statistically noticeable after the mechanism is warmed up. Nevertheless, it is important to understand how the mechanism behaves during a cold start, as this directly impacts an application's startup time. In this subsection we will examine the overhead for a cold start, coming from two different sources:

- The runtime class specialization;
- The cold start of the double factory mechanism.

The evaluation checks the two overheads separately: in the first experiment we only load the classes (using `Class.forName`) to trigger the runtime class specialization, while in the second experiment we instantiate the classes, either directly, using the `new` operator or through the double factory mechanism. In order to evaluate the class specialization, we instrumented the specializing classloader to dump the resulting class files, such that we can compare the specializing classloader to simply loading the specialized variants from the classpath.

For the comparison, we use the `Vector` class described in the previous section. The `Vector` class mixes in 36 traits [**?**] which are translated by the Scala compiler as transitive dependencies of the class. In our experiments, loading the `Vector` class using `Class.forName` transitively loaded another 24 specialized classes for each variant. Instantiating a vector using `new` further loads another 18 classes, mainly specialized trait implementations and internal classes, leading to a total of 42 classes loaded with each specialized variant of `Vector`.

In each experiment we start the virtual machine, start counting the time, load or instantiate `Vector` for all 9 value types in Scala, output the elapsed time and exit. Once a class is loaded, its internal representation in the virtual machine remains cached until its classloader is garbage collected. In order to perform correct benchmarks, we chose to use a virtual machine to load the 9 specialized variants of `Vector` only once, and then restart the virtual machine. We repeated the process 100 times for each measurement.

The first experiment involves loading the class: this can be done either by using the specializing classloader to instantiate a template or by loading the class file dumped from a previous specialization run. We observed a significant difference between cold starting the specializing classloader and warming it up on a different set of classes. This is shown in Table 1.9: cold starting the specialization classloader incurs a slowdown of 153% while warming it up before leads to a 65% slowdown in class loading time.

The second experiment involves instantiating the class, either directly (using the `new` operator) or through the double factory mechanism. Table 1.10 presents the results. The surprising result of this experiment is that the overhead caused by the double factory mechanism is under 4%. As before, most of the time is spent specializing the template to produce the specialized class, which, depending on whether the classloader was used before, can lead to a slowdown between 84% and 144%. It is important to point out this overhead is a one-time cost, and further instantiations of the specialized variants take on the order of tens of milliseconds.

Heap Overhead

In this section we will attempt to bound the heap usage of the double factory mechanism. The double factory mechanism consists of a first level factory, which uses reflection to create second level factories, which, in turn, use the new operator to instantiate load-time specialized classes. This mechanism was imposed in order to avoid the cost of reflection-based instantiation, which we found to be more expensive in terms of overhead. Each second level factory

corresponds to a set of pre-determined type tags, thus instantiating two specialized variants will require two separate second level factories.

The first level factory mechanism keeps a cache of 10^n references pointing to second level factories, which is initially empty and fills up as the different variants are created. The second level factories are completely stateless and only offer a method for each specialized class constructor. Therefore the maximum heap consumption, for a 64 bit system running the HotSpot Virtual Machine, would be 16 bytes for each second level factory and 8 bytes for its cached reference, all times 10^n, assuming all variants are loaded. This means a total of 24×10^n bytes of storage. For a class with a single type parameter, this would mean a heap overhead in the order of hundreds of bytes. Assuming all of spire's specialized classes used arrays and required the two factory mechanism, since most take a single type parameter, it would mean a heap overhead in the order of tens of kilobytes.

However a hidden overhead is also present, consisting of the internal class representations for the second level factories inside the virtual machine. To bound this overhead, we can compare the factories to the classes themselves: for each specialized variant of the class there will be a specialized factory, with a method corresponding to each constructor of the class. The factory will therefore always have a strictly smaller internal representation than the specialized class, leading to at most a doubling of the internal class representation in the virtual machine.

1.7.8 Extending to Other Virtual Machines

In order to asses whether the miniboxing runtime system provides good performance on other virtual machines, we have evaluated it on Graal [**?**]. The Graal Virtual Machine consists of the same interpreter as the HotSpot Virtual Machine but a completely rewritten just-in-time compiler. Since the interpreter is the same, the same type profiles and hotness information is recorded, but the code is compiled using different transformations and heuristics. The results in Table 1.8 exhibit both a much lower variability but also a lower peak performance compared to the C2 compiler in HotSpot (in Table 1.3). With the single exception of `ArrayBuffer`'s `contains` benchmark, the switching runtime support with class loading behaves similarly to specialized code.

1.7.9 Evaluation Remarks

After analyzing the benchmarking results, we believe the miniboxing transformation with type byte switching and classloader duplication provides the most stable results and fulfills our initial goal of providing an alternative encoding for specialization, which produces less bytecode without sacrificing performance. Using the classloader for duplication and switch elimination, the type byte switching does not require forced inlining, making the transformation work without any inlining support from the Scala compiler.

1.8 Related Work

The work by *Sallenave* and *Ducournau* [**?**] shares the same goals as miniboxing: offering unboxed generics without the bytecode explosion. However, the target is different: their Lightweight Generics compiler targets embedded devices and works under a closed world assumption. This allows the compiler to statically analyze the .NET bytecode and conservatively approximate which generic classes will be instantiated at runtime and the type arguments that will be used. This information is used to statically instantiate only the specialized variants that may be used by the program. To further reduce the bytecode size, instantiations are aggregated together into three base representations: `ref`, `word` and `dword`. This significantly reduces the bytecode size and does not require runtime specialization. At the opposite side of the spectrum, miniboxing works under an open-world assumption, and inherits the opportunistic and compatible nature from specialization, which enables it to work under erasure [**?**], without the need for runtime type information. Instead, type bytes are a lightweight and simple mechanism to dispatch operations for encoded value types.

According to *Morrison et al* [**?**] there are three types of polymorphism: *textual polymorphism*, which corresponds to the heterogeneous translation, *uniform polymorphism* which corresponds to the homogeneous translation and *tagged polymorphism* which creates uniform machine code that can handle non-uniform store representations. In the compiler they develop for the *Napier88* language, the generated code uses a tagged polymorphism approach with out-of-band signaling, meaning the type information is not encoded in the values themselves but passed as separate values. Their encoding scheme accommodates surprisingly diverse values: primitives, data structures and abstract types. As opposed to the Napier88 compiler, the miniboxing transformation is restricted to primitives. Nevertheless, it can optimize more using the runtime specialization approach, which eliminates the overhead of tagging. Furthermore, the miniboxing runtime support allows the Java Virtual Machine to aggressively optimize array instructions, which makes bulk storage operations orders of magnitude faster. The initial runtime support implementations presented in §1.5 show that it is not possible to have these optimizations in a purely compiler-level approach, at least not on the current incarnation of the HotSpot Java Virtual Machine.

Fixnums in Lisp [**?**] reserve bits for encoding the type. For example, an implementation may use a 32-bit slot to encode both the type, on the first 5 bits, and the value, on the last 27 bits. We call this in-band type signaling, as the type is encoded in the same memory slot as the value. Although very efficient in terms of space, the fixnum representation has two drawbacks that we avoid in the miniboxing encoding: the ranges of integers and floating point numbers are restricted to only 27 bits, and each operation needs to unpack the type, dispatch the correct routine and pack the value back with its type. This requires a non-negligible amount of work for each operation. Out-of-band types are used in Lua [**?**], where they are implemented using tagged unions in C. Two differences set miniboxing apart: first, fixnums and tagged unions are used in homogeneous translations, whereas the miniboxing technique simplifies heterogeneous translations. Secondly, miniboxing leverages static type information to eliminate

redundant type tags that would be stored in tagged unions. For example, miniboxing uses the static type information that all values in an array are of the same type: in such a case, keeping a tag for each element, as would be done with tagged unions, becomes redundant. Therefore, we consider miniboxing to be an encoding applicable to strongly typed languages, which reduces the bytecode size of heterogeneous translations, whereas fixnums and tagged unions are encodings best applied to dynamically typed languages and homogeneous translations.

The .NET Common Language Runtime [**?**, **?**] was a great inspiration for the specializing classloader. It stores generic templates in the bytecode, and instantiates them in the virtual machine for each type argument used. Two features are crucial in enabling this: the global presence of reified types and the instantiation mechanism in the virtual machine. Contrarily, the Java Virtual Machine does not store representations of the type arguments at runtime [**?**] and re-introducing them globally is very costly [**?**]. Therefore, miniboxing needs to inherit the opportunistic behavior from specialization. On the other hand, the classloading mechanism for template instantiation at runtime is very basic, and not really suited to our needs: it is both slow, since it uses reflection, and does not allow us to modify code that is already loaded from the classpath. Consequently we were forced to impose the double factory mechanism for all classes that extend or mix-in miniboxed parents, creating redundant boilerplate code, imposing a one-time overhead for class instantiation and increasing the heap requirements.

The *Pizza* generics support [**?**] inspired us in the use of traits as the base of the specialized hierarchy, also offering insights into how class loading can be used to specialize code. The mechanism employed by the classloader to support arrays is based on annotations, which mark the bytecode instructions that need to be patched to allow reading an array in conformance with its runtime type. In our case there is no need for patching the bytecode instructions, as miniboxing goes the other way around: it includes all the code variants in the class and then performs a simple constant propagation and dead code elimination to only keep the right instruction. Miniboxing also introduces the double factory mechanism, which pays the reflective instantiation overhead only once, instead of doing it on each class instantiation. The class generation from a template was first presented in the work of *Agesen et al* [**?**].

Around the same time as Pizza, there has been significant research on supporting polymorphism in Java, leading to work such as *GJ* [**?**], *NextGen* [**?**] and the polymorphism translation based on reflective features of *Viroli* [**?**]. NextGen [**?**, **?**, **?**] presents an approach where type parameter-specific operations are placed into snippet methods, which are grouped in wrapper classes, one for each polymorphic instantiation. Wrapper classes, in turn, extend a base class which contains the common functionality independent of the type parameters. It also implements a generated interface which gives the subtyping relation between the specialized classes, also supporting covariance and contravariance for the type parameters. Taking this approach of grouping common functionality in base classes, as specialization does, could reduce code duplication in miniboxed variants, at the cost of duplicating all snippet methods from the parent in the children classes. Since the collections hierarchy in Scala is up to 6

levels deep, the cost of duplicating the same snippet method 6 times outweighs the benefit of reducing local duplication in each class.

The dispatcher objects in miniboxing are specialized and restricted *where clauses* from *PolyJ* [**?**]. Since the methods that operate on primitive values are fixed and known a priori, unlike PolyJ, we can use dispatcher objects and type tags without any change to the virtual machine. Nevertheless it is worth noting that our implementation does pay the price of carrying dispatcher objects in each instance, which PolyJ avoids by implementing virtual machine support for invoking methods in where clauses.

In the context of ML, *Leroy* presented the idea of mixing boxed and unboxed representations of data and described the mechanism to introduce coercions between the two whenever execution passes from monomorphic to polymorphic code or back [**?**]. Miniboxing introduces similar coercions between the boxed and miniboxed representation, whenever the expected type is generic instead of miniboxed. The peephole optimization in miniboxing could be seen as a set of rules similar to the ones given by *Jones et al* in [**?**]. The work on passing explicit type representations in ML [**?**, **?**, **?**, **?**] can also be seen as the base of specialization and also miniboxing. However, since we control rewiring and do it in a conservative fashion, we only use the type tags available, thus miniboxing does not need any mechanism for type argument lifting.

This paper has systematically avoided the problem of name mangling, which has been discussed in the context of Scala [**?**] and more recently of X10 [**?**]. Finally, miniboxing is not limited to classes and methods, but could also be used to reduce bytecode in specialized translations of random code blocks in the program [**?**].

1.9 Conclusions

We described miniboxing, an improved specialization transformation in Scala, which significantly reduces the bytecode generated. Miniboxing consists of the basic encoding (§1.3) and code transformation (§1.4), the runtime support (§1.5) and the specializing classloader (§1.6). Together, these techniques were able to approach the performance of monomorphic and specialized code and obtain speedups of up to 22x over the homogeneous translation (§1.7).

1.10 Introduction

Generics on the Java platform are compiled using the erasure transformation [**?**], which allows them to be fully backward compatible with pre-generics bytecode. Unfortunately, this also means that they only handle by-reference values (objects) and not primitive types. Thus, primitive values such as bytes and integers have to be converted to heap objects each time they interact with generics. This conversion, known as boxing, compromises the execution

performance and increases the heap footprint, forcing Java to lag behind lower-level languages such as C or C++.

The performance drawbacks of erasure are currently being addressed in Project Valhalla[6], an important undertaking led by the Java platform architects and aimed at providing unboxed generics for Java and other JVM languages. The updated bytecode format in Project Valhalla will include the necessary type information to allow load-time class specialization, effectively creating different versions of classes that directly support primitive types. This load-time transformation approach is also employed by the .NET framework [**?**, **?**] in order to implement generics.

Unlike .NET generics, which are always specialized, the current design of Project Valhalla, as of August 2015, makes it an explicit goal to have specialization as an opt-in transformation. This will allow the ecosystem to evolve smoothly from erased to specialized generics, allowing both erased and specialized classes to work side by side. However, in the current early prototype, there are still some restrictions: (1) erased code cannot handle specialized instances in a generic manner and, (2) abstracting over specialized classes using wildcard types [**?**, **?**] pays the cost of boxing primitive types. This is shown in the following example:

```
1  // The Box class is specialized by virtue of its type
2  // parameter T being annotated with the "any" prefix:
3  public class Box<any T> {
4    ...
5    T getValue() { ... }
6  }
7
8  // The getBoxValue method is compiled with erasure
9  // since U is not marked with the "any" prefix:
10 static <U> U getBoxValue(Box<U> box) {
11   return box.getValue();
12 }
13
14 // (1) erased code cannot handle specialized class
15 //      instances (the code will not compile):
16 getBoxValue(new Box<int>(5));
17
18 // (2) abstracting over a specialized class leads to
19 //      boxing the value (any acts as a wildcard type):
20 Box<any> box = new Box<int>(5);
21 System.out.println(box.getValue()); // boxes the value
```

These two code patterns could easily be rewritten to make the code compile and to avoid the overhead of boxing. For the first pattern, adding the `any` prefix to the type parameter `U` of method `getBoxValue` would make it specialized as well, allowing it to handle the incoming argument. In the second pattern, the wildcard, which is equivalent to extending `Box<?>` to value types, could be replaced by the exact type, `Box<int>`, eliminating the overhead of boxing. Yet, in the general case, not all code can be changed at will, due to interactions, backward

[6] The Valhalla Project is still in its infancy, but early prototypes are openly available and the hard goals have been clearly defined in [**?**, **?**, **?**].

compatibility or because it resides in external libraries. Thus, a better solution would be to have erased and specialized generics interoperate, ideally without the overhead.

The Scala programming language, which also compiles to JVM bytecode, has had compile-time specialization for 6 years [**?**, **?**] and currently has three mechanisms for compiling generics: erasure, specialization and a new arrival, miniboxing [**?**]. In Scala, all three generics compilation schemes can be freely mixed:

```
1 // The Mbox class is miniboxed by virtue of the type
2 // parameter annotation (but could be specialized
3 // as well, using @specialized):
4 class Mbox[@miniboxed T](value: T) {
5   def getValue(): T = ...
6 }
7
8 // The getMboxValue method is erased:
9 def getMboxValue[U](mbox: Mbox[U]): U = mbox.getValue()
10
11 // (1) erased code can handle specialized instances:
12 getMboxValue(new Mbox[Int](5))
13 // (2) programmers can abstract over specializations:
14 val mbox: Mbox[_] = new Mbox[Int](5)
15 println(mbox.getValue())
```

Despite the uniform behavior, Scala does pay a hefty price for being able to freely mix code using the three generics compilation schemes: calls between different compilation schemes require boxing primitive values. The reason is that only boxed primitive values are understood by all three transformations. Furthermore, as we will see later on, instantiating a miniboxed (or specialized) class from erased code leads to the erased version being instantiated instead of its miniboxed (or specialized) equivalent, in turn leading to unexpected performance regressions.

In this paper, we show how we completely eliminate the unexpected slowdowns in the miniboxing transformation and, as a side effect, allow programmers to easily and robustly use miniboxing to speed up their programs. The underlying property we are after is that, inside hot loops and performance-sensitive parts of the program, all generic code uses the same compilation scheme, in this case, miniboxing. This way, primitive types are always passed using the same data representation, whether that's the miniboxed encoding (for miniboxing) or the unboxed representation (for specialization).

We show two approaches for harmonizing the compilation scheme across performance-sensitive code:

Issuing actionable performance advisories when compilation schemes do not match, allowing the programmer to harmonize them. For example, when a generic method takes a miniboxed class as a parameter and tries to call methods on it, we automatically generate performance advisories:

```
scala> def getMboxValue[U](mbox: Mbox[U]): U =
    |     mbox.getValue()
<console>:9: warning: The following code could benefit from miniboxing if the type
parameter U of method getMboxValue would be marked as "@miniboxed U":
       mbox.getValue()
            ^
```

Another problem that occurs frequently concerns library evolution: as a new compilation scheme arrives, it is best if all libraries start using it as soon as possible. However, backward compatibility prohibits changing the compilation scheme for the standard library, as it would break old bytecode. In Scala, we had this problem because many of the core language constructs, such as functions and tuples use specialization instead of miniboxing. Similarly, Java has as many as 20 manual specializations for the arity 1 lambda, such as `IntConsumer`, `IntPredicate` and so on. Replacing these by a single specialized functional interface would be desirable, but is realistically impossible. We present a solution for this:

Efficiently bridging the gap between compilation schemes. In the case of miniboxing, which is a compiler plugin, we were not able to change the Scala standard library functions and tuples to use the miniboxing scheme. Instead, we describe the approaches we use to efficiently communicate to the existing library classes, and, where necessary, to replace them by miniboxed equivalents.

With this, the paper makes four key contributions to the Java community and, in the general sense, to the field of compiling object-oriented languages with generics:

- Describing the problems involved in mixing different generics compilation schemes (§1.11);
- Describing a general mechanism for harmonizing the compilation scheme (§1.12);
- Describing the approaches we use to fast-path communication between different generic compilation schemes (§1.13);
- Validating the approach using the miniboxing plugin (§1.14).

The evaluation section (§1.14) shows that warnings not only help avoid performance regressions, but can also guide developers into further improving their program's performance.

1.11 Compilation Schemes for Generics

This section describes the different compilation schemes for generics in Scala. We mainly use Scala for the examples, but the discussion can be applied to Java as well. Differences between Scala specialization and Project Valhalla are pointed out along the way.

1.11.1 Erasure in Scala

The current compilation scheme for generics in both Java and Scala is called erasure, and is the simplest compilation scheme possible for generics. Erasure requires all data, regardless of its type, to be passed in by reference, pointing to heap objects. Let us take a simple example, a generic `identity` method written in Scala:

```
1 def identity[T](t: T): T = t
2 val five = identity(5)
```

When compiled, the bytecode for the method is[7]:

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

As the name suggests, the type parameter `T` was "erased" from the method, leaving it to accept and return `Object`, `T`'s upper bound. The problem with this approach is that values of primitive types, such as integers, need to be transformed into heap objects when passed to generic code, so they are compatible with `Object`.This process, called boxing goes two ways: the argument of method `identity` needs to be boxed while the return value needs to be unboxed back to a primitive type:

```
1 val five = identity(Integer.valueOf(5)).intValue()
```

Boxing primitive types requires heap allocation and garbage collection, both of which degrade program performance. Furthermore, when values are stored in generic classes, such as `Vector[T]`, they need to be stored in the boxed format, thus inflating the heap memory requirements and slowing down execution. In practice, generic methods can be as much as 10 times slower than their monomorphic (primitive) instantiations. This gave rise to a simple and effective idea: specialization.

1.11.2 Specialization

Specialization [**?**, **?**] is the second approach used by the Scala compiler to translate generics and, for methods, is similar to Project Valhalla. It is triggered by the `@specialized` annotation:

```
1 def identity[@specialized T](t: T): T = t
2 val five = identity(5)
```

Based on the annotation, the specialization transformation creates several versions of the `identity` method:

[7] Throughout the paper, we show the source-equivalent of bytecode. The context clarifies whether we are showing source code or compiled bytecode.

```
def identity(t: Object): Object = t
def identity_I(t: int): int = t
def identity_C(t: char): char = t
// ... and another 7 versions of the method
```

Having multiple methods, also called specialized variants or simply specializations of the `identity` method, the compiler can optimize the call to `identity`:

```
val five: int = identity_I(5)
```

This transformation side-steps the need for a heap object allocation, improving the program performance. However, specialization is not without limitations. As we have seen, it creates 10 versions of the method for each type parameter: the reference-based version plus 9 specializations (Scala has the 8 primitive types in Java plus the `Unit` primitive type, which corresponds to Java's `void`). And it gets worse: in general, for N specialized type parameters, it creates 10^N specialized variants, the Cartesian product covering all combinations.

Lacking Project Valhalla's virtual machine support, Scala specialization generates the specialized variants during compilation and stores them as bytecode. This prevents the Scala library from using specialization extensively, since many important classes have one, two or even three type parameters. This led to the next development, the miniboxing transformation.

1.11.3 Miniboxing

Taking a low level perspective, we can observe the fact that all primitive types in the Scala programming language fit within 64 bits. This is the main idea that motivated the miniboxing transformation [**?**]: instead of creating separate versions of the code for each primitive type alone, we can create a single one, which stores 64-bit encoded values, much like C's untagged union. The previous example:

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

Is compiled[8] to the following bytecode:

```
def identity(t: Object): Object = t
def identity_M(..., t: long): long = t
val five: int= minibox2int(identity_M(int2minibox(5)))
```

Alert readers will notice the `minibox2int` and `int2minibox` transformations act exactly like the boxing coercions in the case of erased generics. This is true: the values are being coerced to the miniboxed representation, much like boxing in the case of erasure. Yet, our benchmarks on the Java Virtual Machine platform have shown that the miniboxing conversion cost is completely

[8] In the rest of the paper we assume the miniboxing Scala complier plugin is active unless otherwise noted. For more information on adding the miniboxing plugin to the build please see http://scala-miniboxing.org.

eliminated when just-in-time compiling to native 64-bit code. Further benchmarking has shown that the code matches the performance of specialized code within a 10% slowdown due to coercions [**?**], compared to a 10x slowdown in the case of boxing.

There is an ellipsis in the definition of the `identity_M` method, which stands for what we call a type byte: a byte describing the type encoded in the long integer, allowing operations such as `toString`, `hashCode` or `equals` to be executed correctly on encoded values:

```
1 def string[@miniboxed T](t: T): String = t.toString
```

In order to transform this method, we need to treat the primitive value as its original type (corresponding to `T`) rather than a long integer. To do so, we use the type byte:

```
1 def string(t: Object): String = t.toString
2 def string_M(T_Type: byte, t: long): String =
3                               minibox2string(T_Type, t)
```

Then, when the programmer makes a call to `string`:

```
1 string[Boolean](true)
```

It automatically gets transformed in the compiler pipeline to:

```
1 string_M(BOOL, bool2minibox(true))
```

Knowing the type byte, the `minibox2string` can do its magic: decoding the long integer into a "true" or "false" string, depending on the encoded value. Although seemingly simple, the code transformation to implement the miniboxing transformation is actually rather tricky [**?**, **?**, **?**].

So far, we have only looked at methods, but transforming classes poses even greater challenges.

1.11.4 Class Transformation in Project Valhalla

Project Valhalla takes a straight-forward approach to specialization: classes are duplicated and all previous references to the type parameters are transformed. Given the linked list node class:

```
public class Node<any T> {
  T head;
  Node<T> tail;

  public Node(T head, Node<T> tail) {
    this.head = head;
    this.tail = tail;
  }

  public T head() {
    return this.head;
  }

  public Node<T> tail() {
    return this.tail;
  }
}
```

When specializing the `Node` class for the `int` primitive type, Project Valhalla employs its custom classloader to clone and adapt the bytecode for class `Node`. Among other transformations, it replaces references to `T` by `int` [**?**]:

```
// Node_{T=int} corresponds to Node<int> in the code:
public class Node_{T=int} implements Node_any {
  int head;
  Node_{T=int} tail;
  // ... continued on the next page
```

```
  public Node(int head, Node_{T=int} tail) {
    this.head = head;
    this.tail = tail;
  }

  public int head() {
    return this.head;
  }

  public Node_{T=int} tail() {
    return this.tail;
  }
}
```

The load-time translation produces `Node_{T=int}`, which handles unboxed `int` values. The fact that `Node_{T=int}` is not a subclass of `Node` negatively impacts the interoperation with erasure. Let us take an example method, compiled with erasure:

```
static <U> U getNodeTail(Node<U> spec) {
  return spec.tail();
}
```

The corresponding bytecode is:

```
static Object getNodeTail(Node spec) {
  return spec.tail();
}
```

Since `Node_{T=int}` is not a subclass of `Node`, the `getNodeTail` method cannot handle specialized instantiations of `Node` as arguments, forcing the compiler to reject code patterns such as `getNodeTail(new Node<int>(...))`. This limits the interaction between erased and specialized generics and prevents many useful code patterns from being expressed. In turn, it prolongs the time necessary to adopt specialization: if a single generic library is compiled with erasure, its clients need to use erasure as well, otherwise they would not be able to use that library.

Acknowledging the importance of allowing erased methods to handle specialized instances, a solution was later introduced: In the initial prototype, as of December 2014, the only common parent of classes `Node` and `Node_{T=int}` was `Object`. However, as of August 2015, the translation has been improved to automatically introduce the `Node_any` interface which serves as a common supertype of `Node` and its specializations (e.g. `Node_{T=int}`). Aside from simplifying the translation from JVM languages to Valhalla bytecode, this change also enables programmers to abstract over specialized classes:

```
// Node<any> acts as the Node<?> wildcard, except for
// the fact that it also accepts specialized versions:
Node<any> node = new Node<int>(5, null);
System.out.println(node.getValue());
```

This example is translated to the following bytecode:

```
Node_any node = new Node_{T=int}(5, null);
System.out.println(node.getValue()/* of type Object */)
```

The `Node_any` interface is called an “erased view”, since it enables accessing the specialized class in a uniform, erased-like manner. Since there is no mechanism to return either a reference or a primitive type (corresponding to the `any` wildcard), the call to `getValue()` boxes the value returned. Thus, the call introduces a silent performance regression, which is the cost of interoperability.

The miniboxing translation allows erased, specialized and miniboxed code to freely interact, enabling both code patterns above, at the expense of adding even more boxing operations.

1.11.5 Class Transformation in Miniboxing

Scala specialization [**?**, **?**] introduced a better class translation, which is compatible to erased generics. Miniboxing [**?**] inherited and adapted this scheme, addressing two of its major drawbacks, namely the double fields and broken inheritance. For this reason, we will present the miniboxed class translation scheme directly.

The main challenge of interoperating with erased generics is to preserve the inheritance relation while providing specialized variants of the class, where fields are encoded as miniboxed long integers instead of `Object`s. Let us take the linked list node class again, this time written in Scala:

```
class Node[@miniboxed T](val head:T, val tail:Node[T])
```

The Scala compiler desugars the class to (some aspects omitted):

```
class Node[@miniboxed T](_head: T, _tail: Node[T]) {
  def head: T = this._head       // getter for _head
  def tail: Node[T] = this._tail // getter for _tail
}
```

There are three subtleties in the `Node` translation:

- First, there should be two versions of the class: one where `_head` is miniboxed, called `Node_M` and another one where `_head` is an `Object`, called `Node_L`;
- Then, types like `Node[_]`, which corresponds to Java's wildcard `Node<any>` can be instantiated by both classes, so the two need to share a common interface, the "erased view";
- Finally, this shared interface has to contain the specialized accessors corresponding to both classes (so both classes should implement all the methods).

Given these constraints, miniboxing compiles `Node` to an interface:

```
interface Node {
  def head(): Object    // reference-based accessor
  def head_M(...): long // specialized accessor, which
                        // is not present in the
                        // Valhalla translation
  def tail: Node[T]
}
```

Note that the `tail` method does not have a second version, as it doesn't accept or return primitive values. Then, we have the two specialized variants of class `Node`:

```
class Node_L(_head: Object, _tail: Node) impl Node {
  def head(): Object = this._head
  def head_M(...): long = box2minibox(..., head)
  def tail(): Node[T] = this._tail
}

class Node_M(..., _head: long, _tail: Node) ... {
  def head(): Object = minibox2box(..., head_M(...))
  def head_M(...): long = this._head
  def tail(): Node[T] = this._tail
}
```

As before, the ellipsis corresponds to the type bytes. With this translation, code that instantiates the `Node` class is automatically transformed to use one of the two variants. For example:

```
new Node[Int](4, null)
```

Is automatically transformed to:

```
new Node_M[Int](INT, int2minibox(4), null)
```

And, when `Node` is instantiated with a miniboxed type parameter:

```
def newNode[@miniboxed T](t: T) =
  new Node[T](t, null)
```

The code is translated to:

```
def newNode(t: Object) =
  new Node_L(t, null)
def newNode_M(T_Type: byte, t: long) =
  new Node_M(T_Type, t, null)
```

The translation hints at an optimization that can be done: given a value of type `Node[T]` where `T` is either a primitive or known to be miniboxed, the compiler can call `head_M` instead of `head`, skipping a conversion. The following code:

```
val n = new Node[Int](3, null)
n.head
```

Is translated to:

```
val n = new Node_M(..., 3, null)
n.head_M(...)
```

The rewrite also occurs when the type argument is miniboxed:

```
def getFirst[@miniboxed T](n: Node[T]) = n.head
```

This method is translated to:

```
def getFirst(n: Node): Object =
  n.head           // using reference accessor
def getFirst_M(T_Type: byte, n: Node) =
  n.head_M(T_Type) // using miniboxed accessor
```

At this point, you may be wondering why the `getFirst` bytecode receives a parameter of type `Node` instead of `Node_L`, or, respectively, `Node_M`. The reason is interoperability with erased generics.

1.11.6 Interoperating with Erased Generics

So far, we have seen the following two invariants:

- we call the `head_M` accessor on receivers of type `Node[T]` where `T` is either miniboxed or is a primitive type, optimistically assuming the receiver is an instance of class `Node_M`;
- otherwise, we call the `head` accessor, assuming the receiver is an instance of class `Node_L`.

Unfortunately, interoperating with erased generics violates both invariants. Consider the following method:

```
def newNodeErased[T](head: T) =
  new Node[T](head, null)
```

During the compilation of this method, using to erased generics, the compiler is forced to make a static (compile-time) choice: Which class to instantiate for the `new Node[T]`?

Since `newNodeErased` can be called with both (boxed) primitives and objects, the only valid choice is `Node_L`, which can handle both cases. Contrarily, `Node_M` can't handle references, since object pointers are not directly accessible in the JVM. Thus, we have:

```
def newNodeErased(head: Object) =
  new Node_L(head, null)
```

Which allows the erased generics to invalidate the invariants:

```
val n: Node[Int] = newNodeErased[Int](3)
n.head // n: Node[Int] => call head_M
```

```
val n: Node = newNodeErased(...)    // retuns a Node_L
n.head_M(INT) // assumption: receiver has type Node_M
```

This way, the call to `head_M` occurs on a `Node_L` instance. The symmetric case can also occur, calling `head` on a `Node_M` class. And, what is worse, we can end up with a `Node_L` class storing a primitive value, which means it will be boxed.

Fortunately, by never promising more than the erased view, `Node`, the compilation scheme is robust enough to handle the mix-up. This allows correctly compiling both patterns in the Valhalla example:

```
def getNodeTail[T](t: Node[T]): Node[T] = t.tail
getNodeTail(new Node[Int](5, null))
val node: Node[_] = new Node[Int](5, null)
node.tail()
```

By producing the following bytecode:

```
def getNodeTail(t: Node): Node = t.tail()
getNodeTail(new Node_M(...))      // Node_M impls Node
val node: Node = new Node_M(...) // Node_M impls Node
node.tail() // call through the Node interface
```

Comparing to Valhalla bytecode, the `Node` interface corresponds to Valhalla's `Node_any` interface, the "erased view". Yet, by replacing references to `Node[T]` by the erased view (instead of the `Node_L` class) miniboxing allows better interoperation.

1.12 Performance Advisories

The previous section has shown that, when used globally, miniboxed generics provide two key invariants that ensure primitive values are always passed using the miniboxed (long integer) encoding:

- Instantiations of miniboxed classes use the most specific variant (e.g. a value of type `Node[Int]` has runtime class `Node_M`);
- Methods called on a miniboxed class use the most specific specialization available (e.g. a runtime class `Node_M` always receives calls to the miniboxed `head_M` accessor)

The presence of erasure and wildcard-type abstractions (such as `Node[_]`) leads to violations of these two invariants: the reference variant of a miniboxed class may be instantiated in place of a miniboxed variant or the method called may not be the most specific one available. In both cases, the compilation scheme is resilient, producing correct results, at the expense of performance regressions, caused by boxing primitive types.

There key to avoiding these subtle performance regressions is to intercept the class instantiations and method calls that violate the invariant and report actionable advisories to the users, in the form of compiler warnings. Luckily, all the information necessary to detect invariant violations is available during compilation.

1.12.1 Performance Advisories Overview

Advisories are most commonly triggered by interacting with erased or specialized generics, but can also be caused by technical or design limitations. There are as many as ten different performance advisories implemented in the miniboxing plugin, but in order to focus on the concept, we will only look at the three most common advisories, two of which are caused by the interaction with erased generics. To show exactly how the slowdowns occur, we can take the following piece of code:

```
1 def foo[@miniboxed T](t: T): T = bar(t)
2 def bar[@miniboxed U](u: U): U = baz(u)
3 def baz[@miniboxed V](v: V): V = v
```

The code is compiled to:

```
def foo(t: Object): Object = bar(t)
def bar(u: Object): Object = baz(u)
def baz(v: Object): Object = v
def foo_M(..., t: long): long = bar_M(..., t)
def bar_M(..., u: long): long = baz_M(..., u)
def baz_M(..., v: long): long = v
```

The translation shows that once execution entered the miniboxed path, by calling `foo_M`, it goes through without any boxing, only passing the value in the encoded (miniboxed) representation. Now let's see what happens if the `@miniboxed` annotation is removed:

```
def foo[@miniboxed T](t: T): T = bar(t)
def bar[T](u: U): U = baz(u)
def baz[@miniboxed V](v: V): V = v
```

The bytecode produced is:

```
def foo(t: Object): Object = bar(t)
def bar(u: Object): Object = baz(u)
def baz(v: Object): Object = v
def foo_M(..., t: long): long = box2minibox(bar(minibox2box(t))) // boxing :(
def baz_M(..., v: long): long = v
```

Two problems occur here:

- When method `foo_M` is called, it does not have a miniboxed version of `bar` to call further on, so it calls the erased one;
- When method `bar` is called, although `baz` has a miniboxed version, it cannot be called as the type information was erased.

These two problems correspond to the main two performance advisories: forward and backward warnings. A third one, related to data representation ambiguity, will be shown below.

Forward advisories.

The first advisory (compiler warning) received by the programmer is also called a forward warning:

```
test.scala:7: warning: The method bar would benefit from miniboxing type parameter
 U, since it is instantiated by miniboxed type parameter T of method foo:

     def foo[@miniboxed T](t: T): T = bar(t)
                                      ^
```

This advisory pushes the miniboxed representation from caller to callee when the arguments need to be boxed before being passed.

Backward advisories.

The miniboxing annotation is also propagated from callee to caller:

```
1 test.scala:8: warning: The following code could benefit from miniboxing
  specialization if the type parameter U of method bar would be marked as "@miniboxed
  U" (it would be used to instantiate miniboxed type parameter V of method baz):
2
3       def bar[U](u: U): U = baz(u)
4                             ^
```

Ambiguity advisories.

Scala allows types to abstract over both primitives and objects. For example, wildcard types (known as existentials in Scala) can abstract over any type in the language. `Any` is the top of the Scala type system hierarchy, with two subclasses: `AnyVal` is the superclass of all value types (and thus primitives) while `AnyRef` is the superclass of all reference types, corresponding to Java's `Object`. Therefore, existentials, `Any` and `AnyVal` are not specific enough to pick a primitive or a reference representation. In this case, we issue a warning and box the values:

```
1 test.scala:12: warning: Using the type argument "Any" for the miniboxed type
  parameter T of method foo is not specific enough, as it could mean either a
  primitive or a reference type. Although method foo is miniboxed, it won't benefit
  from specialization:
2           foo[Any](5)
3               ^
```

With these actionable warnings, even a novice programmer, not familiar to the miniboxing transformation, can still achieve the same performance as an expert manually sifting through the generated bytecode. We have several examples where programmers achieved speedups over 2x just by following the miniboxing advisories [**?**, **?**, **?**]. We will now explain the intuition behind generating performance advisories.

1.12.2 Unification: Intuition

The reason we chose to present the "forward", "backward" and "ambiguity" advisories is because, although they are only three of the ten cases, they are the warnings a typical programmer is most likely to encounter. They appear in all cases where a specialized variant of either a method or class needs to be chosen:

- Calling a miniboxed method;
- Instantiating a miniboxed class;
- Calling the method of a miniboxed class;
- Extending a miniboxed class or trait;

The one element common to all these cases is the need to pick the best matching miniboxed variant for the given type arguments. For example, given the method `foo` defined previously, for a call to `foo[Int](4)`, the compiler needs to find the best variant of `foo` and redirect the code to it. In this case, since the type argument of method `foo` is `Int`, which is a primitive type, and since the type parameter `T` in the definition of `foo` is marked as miniboxed, it will pick the `foo_M` variant, which uses the miniboxed representation. This operation is called unification, and we have unified the type parameter of `foo`, namely `T`, and a type argument, `Int`. The unification algorithm is also responsible for issuing advisories.

Let us now focus on a more formal definition.

1.12.3 Unification: Formalization

Let us call the original method or class `O`, with the type parameters `F1` to `Fn` and `VO` the set of specialized variants corresponding to `O`. Each specialized variant `V`∈`VO` corresponds to a mapping from the type parameters to a representation in the set of {miniboxed, reference, erased}. Let us inverse this mapping, to produce another mapping from type parameters and representations to the specialized variants. Let's call it `VS`.

Then the unification algorithm can be reduced to choosing the corresponding `V`∈`VO`, for a term of type `O[T1, .., Tn]`. This can be done following the algorithm in Figure 1.

Let us take an example to illustrate this:

```
1 class C[@miniboxed M, N] // M is mboxed, N is erased
2 class D[L] extends C[L, Int]
```

When deciding which specialized variant of the miniboxed class `C` to use as class `D`'s parent, we have:

- the original class `O` = `C`;
- the type parameters `F1` = `M` and `F2` = `N`;
- the set of variants `VO` = {`C_M`, `C_L`};
- the inverse mapping `VS` = {`M`: miniboxed and `N`: erased → `C_M`, `M`: reference and `N`: erased → `C_L`}

Now, applying the unification algorithm in Figure 1.5 for the type parameter `F1` = `M` coupled with the type argument `T1` = `L`, it issues a forward warning followed by outputting (`M`: reference). Then, applying it to `F2` = `N` and `T2` = `Int`, it issues a backward warning and outputs (`N`: erased). From the two bindings, we obtain the specialized variant `C_L` to be a parent of `D`. Indeed, this is what happens in practice:

```
scala> class C[@miniboxed M, N]
defined class C

scala> class D[L] extends C[L, Int]
<console>:8: warning: The following code could benefit from miniboxing
specialization if the type parameter L of class D would be marked as "@miniboxed L"
(it would be used to instantiate miniboxed type parameter M of class C):
     class D[L] extends C[L, Int]
           ^
<console>:8: warning: The class C would benefit from miniboxing type parameter N,
since it is instantiated by a primitive type:
     class D[L] extends C[L, Int]
           ^
defined class D

scala> classOf[D[_]].getSuperclass
res7: Class[_ >: D[_]] = class C_L
```

Now it is easy to guess where the forward and backward names come from: the direction in which the miniboxing transformation propagates between the type parameter and the type argument.

1.12.4 Unification: Implementation

The performance advisories are tightly coupled with the unification algorithm, which decides the variant that should be used for transforming the code. The processing is done one step at a time, with a type parameter and type argument pair. We will now show some issues that an implementer must be careful about.

Owner chain status.

Since methods and classes in Scala can be nested in any order, we must be careful to propagate the status of the type parameters in the owner chain. In the following example:

```
def a[@miniboxed A] = {
  def b[@miniboxed B] = {
    // need to be aware the representation of
    // type parameters A and B when deciding
    // which variant of C to instantiate:
    new C[A, B]()
  }
  ...
}
```

When deciding which miniboxed variant of class `C` to instantiate, we need to be aware of the nested methods we are located in as we duplicate and specialize the code: if we're in method `b_M` inside method `a_M`, we can rely on values of type `A` and `B` to be miniboxed. Contrarily, if we are in method `b` inside method `a`, values of type `A` and `B` are references.

../graphs/warnings.pdf

Figure 1.5 – The unification algorithm for picking the data representation of a type parameter.

Caching warnings.

Instead of issuing warnings right away, they are being cached and later de-duplicated. The reason is that issuing too many warnings diminishes their value. Aside from the three advisories shown, there are special advisories dealing with the specialization transformation in Scala and certain library constructs that we show in the next section. Thus, we define an ordering of advisory priority and, if multiple warnings are cached, we only issue the most important ones.

Suppressing warnings.

In certain scenarios, programmers are aware of their sub-optimal erased generic code but, due to compatibility requirements with other JVM programs or due to the fact that code lies outside the hot path, they chose not to change it. In these situations, they need to suppress the warnings, because instead of improving visibility, they might obscure other more important performance regressions in the program. However, a coarse-grained approach such as turning off all warnings is not desirable either, as it completely voids the benefit of advisories. For this scenario, the miniboxing transformation provides the `@generic` annotation, which can suppress performance advisories:

```
scala> def zoo[@miniboxed T](t: T) = t
defined method zoo

scala> zoo[Any @generic](3) // no ambiguity warning
res1: Any = 3

scala> def boo[@generic T](t: T) = t
defined method boo

scala> boo[Int](3)          // no backward warning
res2: Int = 3
```

Libraries.

In other cases boxing is caused by the interaction with erased generics from libraries. In this case, the default decision is not to warn, unless the programmer specifically sets the `-P:minibox:warn-all` compiler flag:

```
scala> 3 :: Nil
<console>:8: warning: The method List.:: (located in scala-library.jar) would
benefit from miniboxing type parameter B, since it is instantiated by a primitive
type:
           3 :: Nil
             ^
res0: List[Int] = List(3)
```

As we will see in the benchmarking section (§1.14), the performance advisories allow programmers who are not familiar with the transformation to make the same changes an expert would do.

1.13 Interoperating with Existing Libraries

There is a clear parallel between the manual lambda specializations that are already in the Java Standard Library and thus cannot be eliminated and the specialized constructs in the Scala Standard Library, which cannot be replaced by a compiler plugin. Project Valhalla brings the ability to specialize generics to Java, while miniboxing brings a new compilation scheme for Scala generics. What is common between the two cases is the hard requirement that the new transformations work well with the existing constructs, which use different compilation schemes. This is the problem of interoperating with existing libraries.

In this section we will show how performance regressions occur when miniboxed code interacts with the Scala standard library, which uses either erased generics or the original specialization transformation. To counter these performance regressions, we show three approaches to efficiently bridge the gap between the miniboxing and specialization compilation schemes. Although this section mostly focuses on the interoperation between miniboxing and specialization, the techniques are general and can be applied to Java lambdas and Valhalla as well.

1.13.1 The Interoperation Problem

When interacting with the library from miniboxed code, the programmers forget the fact that library constructs, such as tuples and functions, do not share the same compilation scheme. Thus, they expect the same performance and flexibility as when using miniboxed classes. However, calling specialized code from miniboxed methods and vice-versa is not easy. For example:

```
1 def spec[@specialized T](t: T): String = t.toString
2 def mbox[@miniboxed T](t: T): String = spec(t)
```

The code is translated to:

```
1 def spec(t: Object): String = t.toString
2 def spec_I(t: int): String = Integer(t).toString
3 def spec_J(t: long): String = Long(t).toString
4 ... // other 7 specialized variants
5 def mbox(t: Object): String = spec(t)
6 def mbox_M(T_Type: byte, t: long): String = ...
```

The reference-based `mbox` and `spec` methods can directly call each other, since there is a 1 to 1 correspondence. The problem is that, unlike these two methods, none of the specialized variants have a 1 to 1 correspondence to `mbox_M`. This only leaves the reference-based methods as candidates for the direct interoperation between miniboxing and specialization.

Although it may seem like `mbox_M` could directly invoke `spec_J`, since the argument types match, this would be incorrect, as the value `t` in `mbox_M` can be any primitive type, encoded as a long, whereas `t` in `spec_J` can only be a long integer. Thus, if we were to call `spec_J` from

`mbox_M` passing an encoded boolean, instead of returning either "true" or "false", it would return the encoded value of the boolean.

The `mbox_M` method has one more piece of information: `T_Type`, the type byte describing the encoded primitive. In theory, the miniboxed method could use this type byte to dispatch the right specialized counterpart:

```
def mbox_M(T_Type: byte, t: long): String =
  T_Type match {
    case INT  => spec_I(minibox2int(t))
    case LONG => spec_M(minibox2long(t))
    ...
  }
```

Although this indirect approach seems to work and can easily be automated, it is actually a step in the wrong direction: the miniboxing transformation would be introducing extra overhead without offering the programmer any feedback on how and why this happens. Furthermore, when multiple type parameters are specialized, all 10^N possible combinations would have to be added to the match, making it very large. This is likely to confuse the Java Virtual Machine inlining heuristics, causing severe performance regressions.

It may seem like the other way around would be easier: allowing specialized code to call miniboxed methods without performing a switch. However this is not the case because, having been developed first, specialization is not aware of miniboxing. Thus, when calling miniboxed methods, specialization invokes the reference version, boxing the arguments and unboxing the returned value.

With this in mind, our decision was to go with simplicity and symmetry: the bridge between miniboxing and specialization goes through boxing. To allow transparency, miniboxing issues performance advisories about specialized code that should be miniboxed:

```
scala> def mbox[@miniboxed T](t: T): String = spec(t)
<console>:8: warning: Although the type parameter T of method spec is specialized,
 miniboxing and specialization communicate among themselves by boxing (thus,
 inefficiently) for all classes ...
```

```
 ... other than FunctionX and TupleX. If you want to maximize performance, consider
  switching from specialization to miniboxing:
      def mbox[@miniboxed T](t: T): String = spec(t)
                                             ^
```

This solution works well with most of the code that lies within the programmer's control, including for the case where 3rd party libraries distribute both a specialized and a miniboxed version. However, the one library which cannot have multiple versions and happens to use specialization is the Scala standard library. The two most wide-spread constructs affected by this are Tuples and Functions, both of which are specialized. This makes the following function a worst-case scenario for vanilla miniboxing:

```
1 def tupleMap[@miniboxed T,
2              @miniboxed U](tup: (T, T), f: T => U) =
3   (f(tup._1), f(tup._2))
```

Despite the annotations, with the vanilla miniboxing transformation, all versions of the `tupleMap` method use reference-based tuple accessors and function applications, leading to slow paths irreversibly creeping into miniboxed code. For many applications, this is a no-go, so our task was to eliminate these slowdowns. In the following subsection we present three possible approaches and show where each works best.

1.13.2 Eliminating the Interoperation Overhead

We show three approaches to eliminating the boxing overhead when calling specialized code from miniboxed classes or methods.

Accessors.

The simplest answer to the problem of inter-operating with specialization is to switch on the type byte, as shown previously. To avoid confusing the Java Virtual Machine inlining heuristics, we can extract the operation into a static method, that we call separately. This approach needs to be implemented both for accessors, allowing the specialized values to be extracted directly into the miniboxed encoding and for constructors, allowing miniboxed code to instantiate specialized classes without boxing. This is the approach taken for `Tuple` classes (§1.13.3);

Transforming objects.

The accessors approach allows us to pay a small overhead with each access. This is a good trade-off when the constructs are only accessed a couple of times during their lifetime, which is the case for tuples. In other cases, such as functions, the `apply` method is presumably called many times during the object lifetime, making it worthwhile to completely eliminate the overhead. In this case, a better approach is to replace the `Function` objects by `MiniboxedFunction`s, introducing conversions between them where necessary. This way, the `apply` method exposed by `MiniboxedFunction` can be called directly, and this can compensate for a potentially greater cost of constructing the `MiniboxedFunction` object. This way, switching on the type bytes is done only once, when converting the function, and then amortizes over the function lifetime (§1.13.4);

New API.

In some cases, the API and guarantees are hardcoded into the platform. This is the case for the Scala `Array` class, for which the original miniboxing plugin chose the accessors approach [**?**]. However, a better tradeoff is achieved by defining a new `MbArray` class with a similar API

but different guarantees. This approach will be briefly mentioned in the Arrays subsection (§1.13.5).

The next sections discuss the three methods above.

1.13.3 Tuple Accessors

The Scala programming language offers a very concise and natural syntax for library tuples, allowing users to write `(3,5)` instead of the desugared `new Tuple2[Int, Int](3,5)`. Similarly, it allows programmers to write `(Int, Int)` instead of `Tuple2[Int, Int]`. If we were to introduce miniboxed tuples, we would not be able to use the syntactic sugar to express programs, losing the support of many programmers. Instead, a better choice is to efficiently access specialized Scala tuples.

Although we don't have statistically significant data, our experience suggests that `Tuple` classes have their components accessed only a few times during their life. Therefore, both for compatibility reasons and to avoid costly conversions, we decided to allow the `Tuple` class to remain unchanged, instead focusing on providing accessors and constructors that use the miniboxed encoding.

The optimized tuple accessors

are written by hand and are explicitly given the type byte:

```
1 def tuple1_accessor_1[T](T_Tag: Byte, tp: Tuple1[T]) =
2   T_Tag match {
3     case INT =>
4       // the call to _1 will be rewritten to a call
5       // to the specialized variant _1_I, which
6       // returns the integer in the unboxed format:
7       int2minibox(tp.asInstanceOf[Tuple1[Int]]._1)
8     ...
9   }
```

Once the tuple is cast to a `Tuple1[Int]`, the specialization transformation kicks in and transforms the call to `_1` into a specialized call to `_1_I`, the integer variant. Since the `int2minibox` conversion also takes an unboxed integer, the overhead of boxing is completely eliminated.

The specialized constructors

are motivated by two observations: (1) allocating tuples in the miniboxed code without special support requires boxing and, even worse (2) the tuples created use the reference-based variant of the specialized class, thus voiding the benefits of having added tuple accessors. The code for the tuple constructors is also written by hand and is very similar to the accessor code: it dispatches on the type tags to create tuples of primitive types, which specialization can rewrite to the optimized variants.

Introducing accessors and constructors

is done by the miniboxing plugin when encountering a tuple access followed by a conversion to the miniboxed representation or when the tuple constructor is invoked with all the arguments being transformed from the miniboxed representation to the boxed one. There are two reasons this step needs to be automated:

- By default, programmers do not have access to the type bytes directly, as this would allow them to introduce unsoundness in the type system (they can inspect their representation using miniboxing reflection, but this is outside the scope);
- One of the reasons tuples are useful is their great integration with the language, allowing a very concise syntax. Asking programmers to use anything other than this syntax would be as bad as developing our own, no-syntax-sugar miniboxed tuple.

With these three changes, benchmarks show a 2x speedup when accessing tuples and a 5% slowdown compared to the equivalent code which accesses the tuples directly. The benchmark we used was a tuple quicksort algorithm (§1.14).

With the three elements above, accessors, constructors and the automatic code rewriting we create a direct bridge between specialized tuples and miniboxed classes. Unfortunately, as we've seen before, adding such accessors has to be a carefully-weighted, context-specific decision, so automating it would not provide much benefit. For example, this choice would not be suitable for functions.

1.13.4 Functions

Like tuples, functions in Scala have a concise and natural syntax, which ultimately desugars to one of the `FunctionX` traits, where X is the function arity. For example:

```
1 val f: Int => Int = (x: Int) => x + 1
```

Desugars to:

```
1 val f: Function1[Int, Int] = {
2   class $anon extends Function1[Int, Int] {
3     def apply(x: Int): Int = x + 1
4   }
5   new $anon()
6 }
```

Since `Function` objects are specialized, the code is compiled to:

```
val f: Function1[Int, Int] = {
  class $anon extends Function1_II {
    def apply_II(x: int): int = x + 1
    def apply(x: Object): Object = // call apply_II
  }
  new $anon()
}
```

When interoperating with miniboxed code, functions can only use the reference-based `apply`, introducing performance regressions.

In our early experiments on transforming the Scala collections hierarchy using the miniboxing transformation [**?**], we were proposing an alternative miniboxed function trait, called `MbFunction`, and were performing desugaring by hand. The performance obtained was good, but desugaring by hand was too tedious. Later on, we received a suggestion from Alexandru Nedelcu stating that, since functions in Scala are specialized, we should be able to interface directly, thus benefiting from the desugaring build into Scala without paying for the boxing overhead.

Our initial approach used accessors, but we soon learned that switching on as many as 3 type bytes with each function application incurs a significant overhead. Instead, we decided to re-introduce `MbFunctionX` within the code compiled by the miniboxing plugin, where `X` is the arity and can range between 0 and 2 (Scala includes functions with arities up to 22, but arities above 2 are no longer specialized). Yet, this time the `MbFunctionX` objects would be introduced automatically.

Code transformation.

The miniboxing plugin automatically transforms `FunctionX` to `MbFunctionX`:

- All references to `FunctionX` are converted to `MbFunctionX`;
- Function definitions create `MbFunctionX` instead of `FunctionX`;

For example, the code:

```
def choice[@miniboxed T](seed: Int): (T, T) => T =
  (t1: T, t2: T) => if (seed % 2 == 0) t1 else t2

val function: Int => Int = choice(Random.nextInt)
List((1,2), (3,4), (5,6)).map(function)
```

Is transformed into:

```
def choice(seed: int): MbFunction2 =
  new AbstractMbFunction2_LL {
    def apply(t1: Object, t2: Object) = ...
    val functionX: Function2 = ...
  }
```

```
def choice_M(T_Type: byte, seed: int): MbFunction2 =
 new AbstractMbFunction2_MM {
   def apply_MM(..., t1: long, t2: long) = ...
   val functionX: Function2 = ...
  }

val function: MbFunction2 = choice_M(...)
List((1,2), (3,4), (5,6)).map(function.functionX)
```

Explaining how the code transformation works is beyond the scope of this paper and has been thoroughly studied in previous literature [**?**, **?**]. The result is that, within miniboxed code, only the `MbFunctionX` representation is used. `FunctionX` is only referenced in a limited number of cases:

- When miniboxed code needs to pass a function to pre-miniboxing code (which uses the `FunctionX` representation);
- When miniboxed code receives a function from pre-miniboxing code (using the `FunctionX` representation);
- When a miniboxed class or method extends a pre-miniboxed entity that takes `FunctionX` arguments;
- When an `MbFunctionX` value is assigned to supertypes of `FunctionX`, it needs to be converted;

Conversions

can occur in both directions, from `FunctionX` objects to `MbFunctionX` and back.

Converting `FunctionX` objects to their miniboxed counterparts is done using switches that allow the newly created `MbFunctionX` to directly call the unboxed `apply`, fără boxing:

```
def function0_bridge[R](R_Tag: Byte, f: Function0[R]): MiniboxedFunction0[R] =
  (R_Tag match {
    case INT =>
      val f_cast = f.asInstanceOf[Function0[Int]]
      new MbFunction0[Int] {
        def functionX: Function0[Unit] = f_cast
        def apply(): Int = f_cast.apply()
      }
    ...
  }).asInstanceOf[MiniboxedFunction0[R]]
```

In the above code, `f` is statically known to be of type `Function[Int]`, thanks to the type byte. This allows the code to introduce `f_cast`, which in turn allows the specialization transformation to rewrite the call from the reference-based `apply` to the unboxed `apply_I`. On its side, miniboxing instantiates `MbFunction0_M` instead of `MbFunction0` and moves the code to the specialized `apply_M` method. With these rewrites, the anonymous `MbFunction` instance can call the underlying function without boxing:

```
new MbFunction0_M {
  def T_Type: byte = INT
  // fast path for function application:
  def apply_M(): long = int2minibox(f_cast.apply())
  // fast path for conversion:
  val functionX: Function0 = f_cast
}
```

Converting `MbFunctionX` objects to `FunctionX` easy, since each `MbFunctionX` object contains its `FunctionX` counterpart in the `functionX` field.

By transforming the function representation, we have eliminated the overhead of calling functions completely. Furthermore, using the previous two strategies to minimize the conversion overhead, we enabled function-heavy applications to achieve speedups between 2 and 14x [**?**, **?**].

1.13.5 Arrays

The array transformation [**?**] is beyond the scope of this paper, but we included it as a good example for using performance advisories.

The `Array` bulk storage in Scala makes certain assumptions that are not compatible with miniboxing, leading to performance regressions in some corner cases. To address this limitation, we introduced a new type of array, dubbed `MbArray`, which integrates very well within the miniboxing transformation. However, since the `MbArray` guarantees do not match the ones offered by Scala arrays, we cannot automate the transformation. Instead, we issue performance advisories to inform programmers about `MbArray`:

```
scala> def newArray[@miniboxed T: ClassTag] =
     |   new Array[T](100)
<console>:8: warning: Use MbArray instead of Array to eliminate the need for
ClassTags and benefit from seamless interoperability with the miniboxing
specialization. For more details about MbArrays, please check the following link:
http://scala-miniboxing.org/arrays.html
```

This concludes the three approaches to interoperating with the specialized Scala library.

1.14 Benchmarks

In this section we show three different scenarios where miniboxing has significantly improved performance of user programs. We specifically avoid mentioning benchmarking methodology, as each of the experiments was ran on a different setup. Yet, all three examples show a clear trend: the techniques explained in the paper improve both performance and the programmer experience.

The RRB-Vector

data structure [**?**, **?**] is an improvement over the immutable `Vector`, allowing it to perform well for data parallel operations. Currently, the immutable `Vector` collection in the Scala library offers very good asymptotic performance over a wide range of sequential operations, but fails to scale well for data parallel operations. The problem is the overhead of merging the partial results obtained in parallel, due to the rigid Radix-Balanced Tree, the `Vector`'s underlying structure. Contrarily, the `RRB-Vector` structure uses Relaxed Radix-Balanced (RRB) Trees, which allow merges to occur in effectively constant time while preserving the sequential operation performance. This enables the `RRB-Vector` to scale linearly with the number of cores when executing data parallel operations. Thanks to its improved performance, the `RRB-Vector` data structure is slated to replace the `Vector` implementation in the Scala library in a future release.

The original `RRB-Vector` implementation used erased generics. To show that performance advisories can indeed guide developers into improving performance, we asked a Scala developer who was not familiar with the `RRB-Vector` code base to switch the compilation scheme to miniboxing. Before handing in the code, we removed the parallel execution support [**?**], reducing the code base by 30%. Then, the developer compiled the code with the miniboxing plugin, which produced 28 distinct warnings. These warnings guided the addition of `@miniboxed` annotations where necessary and the introduction of `MbArray` objects instead of Scala arrays. By following the performance advisories, in less than 30 minutes of work, our developer managed to improve the performance of the `RRB-Vector` operations by up to 3x. A counter-intuitive effect was that it took three rounds of compiling and addressing the warnings before the improvement was visible: each iteration introduced more `@miniboxed` annotations, in turn triggering new warnings, as new methods could benefit from the annotation.

Table 1.11 shows the performance improvements measured on four key operations of the `RRB-Vector`: creating the structure element by element using a builder and invoking bulk data operations: `map`, `fold` and `reverse`. The ScalaMeter framework [**?**] was used as a benchmarking harness on a quad-core Intel Core i7-4600U processor running at 2.10GHz with 12GB of RAM, on OpenJDK7.

The numbers show speedups between 1.9 and 3x for the `builder`, `map` and `fold` benchmarks. This can be explained by the fact that, in the erased version, each element required at least a boxing operation, and thus a heap allocation. On the other hand, the `reverse` operation does not require any boxing so there is no speedup achieved. Nevertheless, introducing the miniboxing transformation does not lead to significant slowdowns.

If we consider the `RRB-Vector` development time, which took four months of work and resulted in ~3K lines of source code, the performance advisories issued by the miniboxing plugin allowed a new developer, with no knowledge of the code base, to deploy the miniboxing transformation in a negligible period of 30 minutes, producing speedups of up to 3x.

Image processing.

Performance advisories can be used to improve the performance of Scala programs without any previous knowledge of how the transformation works. This was shown at the PNWScala 2014 developer conference [**?**], where Vlad Ureche presented how the miniboxing plugin guides the programmer into improving the performance of a mock-up image processing library by as much as 4x [**?**]. The presentation was recorded and the performance numbers are included in Table 1.12 for quick reference.

Tuple accessors

have been tested by implementing a tuple sorting benchmark using a generic quicksort algorithm. Table 1.13 shows the results for sorting 1 million tuples based on their first element. We used different transformations for the generic quicksort algorithm: first, we benchmarked the erased generics performance, which, as expected, were slow. Surprisingly, the miniboxed version without tuple support was even worse, 7% slower than erased generics. Then, adding tuple accessor support to the miniboxing transformation improved the performance by 2x, making it comparable to the original specialization transformation and to the monomorphic (non-generic, hand specialized) version of the quicksort algorithm.

1.15 Related Work

The most significant related work lies in the area of run-time profilers which can offer feedback at the language level. We would like to point the work of *St-Amour* on optimization feedback [**?**] and feature-based profiling [**?**]. Profiling has existed for a long time at lower levels, such as at the Java Virtual Machine level, with profilers such as YourKit [**?**] or the Java VisualVM [**?**] or the x86 assembly, with processor hardware counters.

The area of opportunistic optimizations has seen an enormous growth thanks to dynamic languages such as JavaScript, Python and Ruby, which require shape analysis and optimistic assumptions on the object format to maximize execution speed. We would like to highlight the work of Mozilla on their *Monkey JavaScript VMs [**?**], Google's V8 JavaScript VM and the PyPy Python virtual machine [**?**, **?**]. While this is just a short list of highlights, the Truffle compiler [**?**, **?**] is now a general approach to writing interpreters that make optimistic assumptions, allowing maximum performance to be achieved by partially evaluating the interpreter for the program at hand, essentially obtaining a compiled program thanks to the first Futamura projection [**?**].

In the area of data representation, this work assumes familiarity with specialization [**?**] and miniboxing [**?**, **?**]. The program transformation which enables the functions to be transformed into miniboxed functions is thoroughly discussed in [**?**, **?**]. There has been previous work on miniboxing Scala collections [**?**] and on unifying specialization and reified types [**?**]. We have also seen a revived interest in specialization in the Java community, thanks to project Valhalla,

which aims at providing specialization and value class support at the virtual machine level [**?**, **?**]. In the Java 8 Micro Edition functions are also represented differently [**?**].

1.16 Conclusion

This paper shows several approaches to allowing different generics compilations schemes to interoperate without incurring performance regressions:

- Harmonizing the generics compilation scheme thanks to actionable performance advisories;
- Bridging the gap between library constructs that use different generics compilation schemes, specifically:
  - The accessor approach;
  - The replacement approach;
  - The advisory-based approach.

The implementation results are validated using the miniboxing plugin, which automates the approaches described, showing performance improvements between 2x and 4x.

Acknowledgements

The authors are grateful to the following people who motivated the development of the features described in the paper: Alexandru Nedelcu, Aymeric Genet and Aggelos Biboudis (Functions), Philip Stutz, Stu Hood, Iulian Dragos and Rex Kerr (warnings), Julien Truffaut (tuple accessors). We are thankful to Vincent St-Amour for the ground-breaking work on program optimization advisories.

We would like to thank Brian Goetz, Maurizio Cimadamore and the PPPJ reviewers for their helpful comments and suggestions.

	ArrayBuffer.append		ArrayBuffer.reverse		ArrayBuffer.contains	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	50.1	48.0	20.4	21.5	1580.1	3628.8
mb. switch	30.9	35.5	2.5	15.1	161.5	554.3
mb. dispatch	16.5	58.2	2.1	26.5	160.7	2551.6
mb. switch + LS	**15.6**	**14.8**	**2.5**	**2.4**	**159.9**	**161.7**
mb. dispatch + LS	**15.1**	**15.9**	**2.0**	**2.7**	**161.8**	**161.3**
specialization	39.7	38.5	2.0	2.4	155.8	156.3
monomorphic	16.2	N/A	2.1	N/A	157.7	N/A
	List creation		List.hashCode		List.contains	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi Context
generic	16.7	1841	22.1	20.4	1739.5	2472.4
mb. switch	11.4	11.7	18.3	18.8	1438.2	1443.2
mb. dispatch	11.4	11.5	15.6	21.0	1369.1	1753.2
mb. switch + LS	**11.5**	**11.6**	**16.2**	**16.1**	**1434.9**	**1446.3**
mb. dispatch + LS	**12.1**	**12.7**	**16.1**	**15.3**	**1364.2**	**1325.9**
specialization	11.4	11.4	14.5	36.4	1341.0	1359.2
monomorphic	10.2	N/A	13.3	N/A	1172.0	N/A

Table 1.3 – Benchmark running times. The benchmarking setup is presented in §1.7.2 and the targets are presented in §1.7.3. The time is measured in milliseconds.

	ArrayBuffer			List		
generic	4.6	2.2	367.0	1.4	**0.2**	16.6
mb. switch + LS	**1.6**	**0.3**	**25.0**	**0.8**	**1.3**	**4.2**
mb. dispatch + LS	2.5	0.7	88.9	1.1	1.5	7.3
specialization	4.3	0.5	30.7	0.6	1.9	2.2
monomorphic	1.0	0.2	12.7	0.4	1.2	2.2

Table 1.4 – Running time for the benchmarks in the HotSpot Java Virtual Machine interpreter. The time is measured in seconds as instead of milliseconds as in the other tables. "Single context" and "Multi context" have similar results.

	erasure	dispatch	switch	spec.
ArrayBuffer	4.4	19.5	24.5	57.6
ArrayBuffer factory	–	+ 9.0	+ 8.5	–
ListNode	3.1	10.9	11.5	45.0
ListNode factory	–	+ 8.7	+ 8.3	–

Table 1.5 – Bytecode generated by different translations, in kilobytes. Factories add extra bytecode for the double factory mechanism. "spec." stands for specialization.

	bytecode size (KB)	classes
Spire - specialized (current)	13476	2545
Spire - miniboxed	4820	1807
Spire - generic	3936	1530

Table 1.6 – Bytecode generated by using specialization, miniboxing and leaving generic code in the Spire numeric abstractions library.

	bytecode size (KB)	classes
Vector - specialized	5691	1434
Vector - miniboxed	1210	435
Vector - generic (current)	715	223

Table 1.7 – Bytecode generated by using specialization, miniboxing and leaving generic code on the Scala collection library slice around Vector.

	`ArrayBuffer.append`		`ArrayBuffer.reverse`		`ArrayBuffer.cont`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi
generic	78.3	52.3	3.2	20.3	607.6	31
mb. switch	27.6	×	7.4	×	844.4	
mb. dispatch	27.0	34.8	3.2	10.8	844.7	96
mb. switch + LS	**22.2**	**14.3**	**3.8**	**2.9**	**725.4**	**72**
mb. dispatch + LS	**32.9**	**26.4**	**3.4**	**4.0**	**844.6**	**84**
specialization	21.7	13.4	3.5	2.7	488.7	48
monomorphic	19.8	N/A	3.1	N/A	490.4	N
	`List` creation		`List.hashCode`		`List.contain`	
	Single Context	Multi Context	Single Context	Multi Context	Single Context	Multi
generic	32.6	23.3	13.4	13.6	1846.5	21
mb. switch	23.7	18.0	11.7	10.9	1420.8	14
mb. dispatch	20.9	18.3	12.4	11.4	1359.3	14
mb. switch + LS	**23.2**	**17.1**	**12.2**	**10.5**	**1414.8**	**14**
mb. dispatch + LS	**25.0**	**18.3**	**12.1**	**10.5**	**1390.6**	**14**
specializare	21.7	16.9	12.4	10.6	1463.5	14
monomorphic	19.6	N/A	11.7	N/A	1249.2	N

Table 1.8 – Running times on the Graal Virtual Machine. "×" marks benchmarks for which the bytecode generated crashed the Graal just-in-time compiler. The time is measured in milliseconds.

	time in ms	classes
classpath - just load	182 ± 5	9 × 25 = 225
classloader - warmed up	300 ± 4	225
classloader - cold start	461 ± 9	225

Table 1.9 – Loading time (classpath) and time for cloning and specialization (classloader) for the 9 specialized variants of Vector and their transitive dependencies.

	time in ms	classes
classpath - new	258 ± 5	9 × 42 = 378
classpath - factory	268 ± 6	378
classloader - factory - warm	488 ±10	378
classloader - factory - cold	655 ± 9	378

Table 1.10 – Instantiation time for the 9 specialized variants of Vector and their transitive dependencies.

Benchmark	Generic	Miniboxed
Builder	161.61 s	53.56 s
Map	98.43 s	49.38 s
Fold	87.98 s	46.14 s
Reverse	27.97 s	33.84 s

Table 1.11 – RRB-Vector operations for 5M elements.

Benchmark	Generic	Minibox some advisorie heeded	Miniboxed all advisories heeded
1st run	4192 ms	3082 ms	1346 ms
2nd run	4957 ms	2998 ms	1187 ms
3rd run	4755 ms	3017 ms	1178 ms
4th run	3969 ms	2535 ms	1094 ms
5th run	4073 ms	2615 ms	1163 ms

Table 1.12 – Speedups based on performance advisories, PNWScala

Transformation	Running time
Generic	684.4 ms
Miniboxed (no tuple support)	726.8 ms
Miniboxed (with tuple support)	323.2 ms
Specialized	322.5 ms
Monomorphic	318.1 ms

Table 1.13 – Sorting 1M tuples using quicksort.

2 Late Data Layout

2.1 Introduction

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

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

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

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

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

chosen based on the interaction with the other language features and the necessary coercions between representations, such as `box` and `unbox`, are added to the code.

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

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

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

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

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

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

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

- We survey the existing approaches to data representation transformation, show their limitations and explore the additional features required (§3.2 and §2.3);
- We show the Late Data Layout (LDL) representation transformation mechanism, which does not impose the semantics of alternative representations and coercions (§2.4) and reason intuitively about its properties (§2.5);
- We validate the mechanism by implementing three language features as Scala compiler extensions using the LDL transformation: value classes[1], specialization using the miniboxing representation[2] and a simple staging mechanism[3] (§2.6). For each of these use cases, we describe the implementation in detail, we compare it to the existing transformations in terms of code size and complexity, we evaluate the resulting programs in terms of performance and finally show the specific extensions we added to the LDL mechanism to support each use case.

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

Key Insights

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

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

Generality. The annotations can be introduced either automatically, by the compiler, based on the interactions of different language features, or manually, by programmers. This provides the flexibility necessary to capture a wide variety of transformations: some of them work

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

automatically, like unboxing primitive types and value classes, whereas others, like staging, require manual annotation, corresponding to domain-specific knowledge.

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

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

This flexibility of annotating individual values with their alternative representation is in sharp contrast to state of the art data representation transformations [**?**, **?**]. These transformations consider the unboxed representation as always desirable and hardcode the semantics of erased generics into their transformation rules. Section §2.3.3 shows that being able to selectively annotate the values that use a different representation is crucial to implementing transformations in object-oriented languages. This flexibility is also fundamental to multi-stage programming, where the choice of execution stage has to be done for each individual value.

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

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

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

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

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

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

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

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

2.2 Data Representation Transformations

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

In the rest of the paper we consider the integer concept to be boxed by default and represent it by `Int`. The goal of the transformations is to use the unboxed integer, `int`, whenever possible. Unless otherwise specified, all generic classes are assumed to be compiled to erased

homogeneous low-level code. Finally, to improve readability, we place annotations in front types (e.g. `@unboxed Int`) instead of after (e.g. `Int @unboxed`), as the Scala syntax requires.

2.2.1 Naive Transformations

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

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

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

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

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

2.2.2 Eager (Syntax-driven) Transformations

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

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

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

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

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

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

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

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

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

2.2.3 Peephole Optimization For Eager Transformations

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

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

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

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

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

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

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

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

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

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

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

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

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

Block optimizations. Let us take one final example:

```
1 def bar(): Boolean = {
2   foo(..., ...)
3   true
4 }
```

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

```
1 def bar(): Boolean = {
2   box(foo(unbox(...), unbox(...)))
3   true
4 }
```

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

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

2.2.4 Type-driven Transformations

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

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

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

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

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

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

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

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

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

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

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

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

2.3 Object-Oriented Data Representation

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

2.3.1 Subtyping

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

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

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

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

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

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

2.3.2 Virtual Method Calls

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

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

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

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

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

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

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

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

2.3.3 Selectivity

We argue that selectivity should be built into data representation transformations as a first-class concern, allowing the programmer or the compiler to individually pick the values that will use alternative representations. Most state-of-the-art data representation transformations

make the assumption that all values that can use an alternative representation should use it. However, we identified several cases that invalidate this assumption:

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

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

```
1 @value class D(val x: Int)
2 class E[T] {
3   def id(t: T) = println("boo")
4 }
5 class F extends E[D] {
6   override def id(d: D) = println("ok")
7 }
```

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

```
1 class F extends E[D] {
2   override def id(d: Object) = id(unbox(d))
3   def id(d: int) = println("ok")
4 }
```

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

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

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

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

Summarizing §3.2 and §2.3, we note that an ideal data representation transformation should be smart about introducing coercions, should account for object orientation and should allow for selective coercions. The next section presents exactly that - a general, consistent, optimal, selective and object-oriented data representation transformation.

2.4 Late Data Layout

This section presents an approach to unifying data representation transformations under a general, consistent, optimal and selective mechanism: the Late Data Layout. We start with an overview (§2.4.1) and then present the three phases of the mechanism (§2.4.2-2.4.4), followed by their properties (§2.5).

2.4.1 Overview

The type-driven data representation transformation (§2.2.4) has shown that coercions can be guided by the type system. Still, this approach was limited by the fact that high-level concepts have to injectively map into low-level representations, which is not always the case. Furthermore, as we will see in this section, local type inference [**?**] is the key to optimally introducing coercions in a type-driven transformation.

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

representation (the COMMIT phase). Using annotated types allows high-level concepts to map injectively to alternative representations, enabling type-driven transformations.

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

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

- isomorphism of the representations:
  `box(unbox(t)) = t` and `unbox(box(t)) = t`;

- purity of the coercions: coercions between representations should not produce any side-effects.

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

Throughout the section we use the following example:

```
1 def fact(n: Int): Int =
2   if (n <= 1)
3     1
4   else
5     n * fact(n - 1)
```

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

```
1 def fact(n: Int): Int =
2   if (n.<=(1))
3     1
4   else
5     n.*(fact(n.-(1)))
```

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

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

2.4.2 The INJECT Phase

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

The INJECT phase transforms the running example to:

```
1 def fact(n: @unboxed Int): @unboxed Int =
2   if (n.<=(1: @unboxed Int))
3     (1: @unboxed Int)
4   else
5     n.*(fact(n.-(1: @unboxed Int)))
```

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

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

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

The transformation mechanism injects an annotation checker that allows the different representations to be compatible during the INJECT phase. This is done on purpose in the LDL mechanism, to allow the delayed introduction of coercions. Should annotated types be incompatible in the INJECT phase, the AST would become type-inconsistent, requiring the

introduction of coercions to regain consistency. But there is a big win in being able to manipulate the tree with annotations but without coercions: for miniboxing, methods can be redirected to "specialized" variants without worrying about coercions, while for value classes and primitive unboxing, bridge methods can forward to their target without explicitly coercing the arguments.

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

2.4.3 The COERCE Phase

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

The COERCE phase transforms the abstract syntax tree in two steps: (1) in the annotation checker, the different representations become incompatible, thus invalidating the current AST and (2) the COERCE phase type-checks the AST and introduces coercions where necessary. The coercions are introduced based on the representation mismatches revealed by the local type inference (§2.4.3): when a certain representation is required but a different one is passed, a coercion is introduced (§2.4.3). Since the names have been resolved and the tree has been type-checked before, type-checking the tree again will only be responsible for inserting coercions (given that the type checker is indeed idempotent). Finally, object-oriented features of the language need to be taken into account (§2.4.3).

Local Type Inference

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

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

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

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

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

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

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

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

Placing Coercions

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

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

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

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

Propagating expected types delays the introducing the coercions until a node with a fixed type is encountered, such as the value `x` in the previous example, and the expected type requires a different representation. This sinks coercions as deep in the AST as possible, side-stepping the need for a peephole optimization (§2.2.3) and making the coercion insertion optimal. Optimality is further discussed in §2.5.3.

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

Object-Oriented Aspects

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

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

```
1 def fact(n: @unboxed Int): @unboxed Int =
2   if (n._<=_(1: @unboxed))
3     (1: @unboxed)
4   else
5     n._*_(fact(n._-_(1: @unboxed)))
```

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

2.4.4 The COMMIT Phase

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

```
1 def fact(n: int): int =
2   if (n._<=_(1))
3     1
4   else
5     n._*_(fact(n._-_(1)))
```

The COMMIT phase is heavily dependent on the transformation at hand when updating the symbol signatures and the AST. For certain transformations, it can go beyond replacing coercion markers by actual operations: unboxing multiple-parameter value classes requires creating multiple fields and populating them. Yet, the AST transformations have local scopes and are always triggered either by a coercion marker, an annotated type in the node or a library method that carries special semantics for the given transformation. For example, in the staging plugin, the method `compile[T](expr: @staged T): T` has the special meaning that a staged expression needs to be compiled to optimized code and executed. It is redirected by the staging plugin from identity (the default implementation, in the case staging is turned off) to a special implementation that generates the code, compiles it and invokes the result. The Validation section of the paper (§2.6) describes the rules of the COMMIT phases for each of the three extensions we developed using the Late Data Layout mechanism.

2.5 Transformation Properties

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

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

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

2.5.1 Consistency

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

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

2.5.2 Selectivity

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

2.5.3 Optimality

Experience with the LDL mechanism reveals an interesting fact: Thanks to the expected type propagation in local type inference, representation constraints are propagated deeper in the AST and, in certain branches or expressions, the coercions are elided completely, when the expected representation matches the actual one. This leads us to think that, for any given execution trace of the input program, the LDL mechanism minimizes the number of coercions executed. While we do not formally prove this property, we give an intuitive explanation of why it occurs. It should be noted that the minimization is done modulo the annotations introduced by the INJECT phase, that dictate which values are unboxed and that can potentially be suboptimal (§2.3.3).

Revisiting the behavior of `if` nodes and blocks, described in Section 2.4.3, we can partition the AST nodes into `opaque` and `transparent`. Opaque AST nodes have a fixed type, which is not influenced by the expected type of the outer expressions. For example, a constant literal `3` is an integer regardless of the expected type. Transparent nodes, on the other hand, adapt to the expected type by further constraining their children AST nodes, as the `if` expression does.

This binary classification does not capture the full wealth of features in Scala's type checker, such as implicit conversions, overloads and polymorphic nodes. However, these are typically resolved during the initial program type-checking phase, in the compiler frontend, and do not influence LDL-based transformations.

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

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

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

```
1 def baz(t1: @unboxed T, t2: @unboxed T, t3: T, c1: Boolean, c2: Boolean): @unboxed
  T =
2   if (c1)
3     t1
4   else
5     if (c2)
6       t2
7     else
8       unbox(t3)
```

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

```
1 def buz(t1: T, t2: T, c: Boolean): @unboxed T =
2   if (c)
3     unbox(t1)
4   else
5     unbox(t2)
```

Since constraints are sunk to the bottom of the tree, they may be duplicated several times for nodes such as conditionals and pattern matches. Therefore, the total number of coercions introduced in the tree is not minimum, in our example being 2, instead of 1, which corresponds to coercing the `if` expression. Still, given any execution trace in the program, the total coercions executed is minimum, in our example, just 1. Note that coercions may be further reduced or increased by changing the output of the INJECT phase (§2.3.3). Also, naively implementing the COMMIT phase can introduce to redundant coercions. Unfortunately, it is impossible to reason about the INJECT and COMMIT phases in a general setting, as they are specific to each transformation.

Arguably, sinking coercions could potentially place them inside hot loops. In Scala, since `for` loops are desugared to method calls, the only two mechanisms for low-level looping are `while` loops and tail-recursive calls. Both `while` loops and method calls are opaque nodes in the AST and do not propagate expected types. Therefore, for the Scala ASTs, we do not expect the LDL mechanism to sink coercions inside hot loops. Still, coercions may be introduced in hot loops based on the annotations introduced by the INJECT phase for loop-local values, which may require coercing (§2.3.3).

2.6 Validation and Evaluation

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

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

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

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

2.6.1 Scala Compiler Plug-ins

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

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

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

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

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

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

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

In Scala, the type checker consists of two parts:

1. a typing judgement, which assigns a type to each AST node and

2. an adaptation routine, which transforms AST nodes so their type matches the expected one.

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

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

We assume the methods have the following signatures:

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

In the type-checking algorithm, the adaptation routine (2) is only triggered if the type of the current tree, as decided by the typing judgement (1), does not conform to the expected type. As a result, only opaque nodes (§2.5.3) reach the adaptation routine. For example, the typing judgement for an `if` expression will propagate the expected type to the branches, leading to each individual branch conforming or being adapted to conform. This makes the conditional itself conform, therefore bypassing adaptation.

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

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

Thanks to the annotation checker, when a node is type-checked expecting a super type, it is automatically boxed. This occurs because, as discussed in §2.3.1, super types of an unboxed type cannot themselves be unboxed. Along with the method call transformation, the super type boxing forms the LDL support for the object-orientated features in the Scala programming language.

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

2.6.2 Case Study 1: Value Classes

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

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

```
1 @value class Meter(val x: Double) {
2   def +(other: Meter) = new Meter(x + other.x)
3   def <=(other: Meter) = x <= other.x
4 }
5 def report(m: Meter) = {
6   if (m.<=(new Meter(9000)))
7     println(m.toString)
8 }
```

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

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

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

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

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

The ADDEXT phase makes several changes to the tree: it adds standard `hashCode` and `equals` implementations for value classes, it transforms value class methods into extensions and finally adds redirects from the value class to the extension methods in the companion object. The extension methods are later used by the COERCE phase, which redirects method calls as described in §2.4.3. The result is:

```
1  @value class Meter(val x: Double) {
2    def +(other: Meter) = Meter.+(this, other)
3    def <=(other: Meter) = Meter.<=(this, other)
4    ... // hashCode, equals redirections
5  }
6  object Meter {
7    def +(self: Meter, other: Meter) =
8      new Meter(self.x + other.x)
9    def <=(self: Meter, other: Meter) =
10     self.x <= other.x
11   ... // hashCode, equals extension methods
12 }
13 def report(m: Meter) = ...
```

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

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

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

Another responsibility of the INJECT phase is the creation of bridge methods (§2.3.3). If a method that has value class parameters overrides a generic method, INJECT creates a corresponding bridge:

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

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

The COERCE phase follows the pattern established in §2.4, making `@unboxed` types incompatible with their non-annotated counterparts and inserting `box` and `unbox` markers in case of representation mismatches. The coerce phase also redirects to extension methods where possible. For our running example, the following code is produced:

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

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

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

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

The COMMIT phase transforms our example to:

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

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

Evaluation

We evaluate the plugins on three metrics:

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

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

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

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

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

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

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

```
1 ::Benchmark FFT.Scala Complex::
2 Parameters(data size = 2^ -> 4): 11.9418295
3
4 ::Benchmark FFT.Valium Complex::
5 Parameters(data size = 2^ -> 4): 11.8187571
```

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

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

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

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

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

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

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

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

2.6.3 Case Study 2: Miniboxing

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

It is worth mentioning that miniboxing exploits all the flexibility available in the LDL mechanism: in the last version it features 2 alternative encodings (miniboxing to `Long` or `Double`), the alternative representation mapping is not injective, since all miniboxed type parameters map to either `Long` or `Double`, the selectivity is used to generate bridge methods for similar reasons to those presented in §2.3.3 and the compatibility between annotated and non-annotated types in the INJECT phase is used to easily redirect method calls from miniboxed methods to their specialized variants.

Evaluation

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

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

- 3 rules for coercions (`minibox2box`, `box2minibox`, `minibox2minibox`);
- 4 rules for redirecting methods inherited from `Any`, such as `toString` - triggered by coercions;

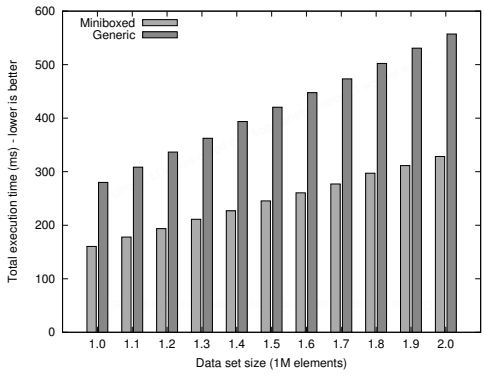


Figure 2.1 – Least squares method using linked lists

- 4 rules for optimizing arrays - triggered by array operations (`a.{apply,update,length}` and `new T[]`);

- 1 extra rule for optimizing the function representation.

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

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

- Before (29th of October 2014): 2285 LOC (out of which approximately 500 LOC in the peephole optimization)

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

Runtime performance. Since the miniboxing plugin has been around for some time, its runtime performance has been thoroughly benchmarked [**?**]. The most recent result is a benchmark on a slice of the Scala collections library [**?**] centered around the linked list collection. The benchmark consists of running the least squares method for fitting data points on several input sizes. The results, summarized in Figure 2.1, show a 45% speedup produced by using the miniboxing transformation. It should be noted that Scala collections are notoriously hard to transform, since they use many advanced features of the language, such as type classes, higher-kinded types and anonymous and nested classes. Indeed, we also tried to run the benchmark with the current specialization transformation in the Scala compiler [**?**], but the results were disappointing: due to technical difficulties, the specialized linked list was slower than the generic one.

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

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

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

These additions are described on the miniboxing website [**?**].

2.6.4 Case Study 3: Staging

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

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

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

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

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

accumulate the operations necessary to produce `b`$^{\texttt{e}}$ for a variable base `b` and a fixed exponent `e`.

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

```
1  function: compiling the following code:
2  ************************************
3    (arg: Double) => {
4      val x0: Double = arg * arg
5      val x1: Double = x0 * x0
6      val x2: Double = arg * x1
7      x2: Double
8    }
9  ************************************
10 3.0^5 = 243.0
11 4.0^5 = 1024.0
```

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

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

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

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

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

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

Evaluation

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

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

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

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

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

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

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

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

2.7 Related Work

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

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

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

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

Shao further extends *Leroy*'s work [**?**, **?**] by presenting a more efficient representation, at the expense of carrying explicit type representations [**?**, **?**]. *Minamide* further refines the transformation and is able to formally prove that the transformed code has the same time complexity as the original program [**?**]. Tracking value representation in types has been presented and extended to continuation-passing style [**?**] by *Thiemann* in [**?**]. Two pieces of information are tracked in a lattice: whether the value corresponding to the type is used at all (otherwise its representation can be ignored - called "Don't care polymorphism" and equivalent to our `oblivious` relation between AST nodes) and whether a certain representation is required.

This information is used in a type inference algorithm which can elide coercions when the parameters are discarded or when a method call is in tail position, namely it doesn't need to box the result only to have the caller unbox it. It should be noted that the coercions operate on a continuation-passing-style intermediary representation.

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

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

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

Specialization for generics is a technique aimed at eliminating boxing deep inside generic classes. Specialization has been implemented in Scala [**?**, **?**] and has been improved by miniboxing [**?**, **?**]. Specialization and macros have been combined to produce a mechanism for

ad-hoc specialization of code in Scala [**?**]. The .NET CLR automatically specializes all generics, thanks to its bytecode metadata and reified types [**?**].

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

Multi-stage programming (also called staging) [**?**] requires lifting certain expressions in the program to a reified representation. Staging can be implemented using macros [**?**, **?**, **?**], or using specialized compiler extensions [**?**]. One of the applications is removing the abstraction overhead of high-level and embedded domain specific languages. Indeed, staging was successfully used to optimize and re-target domain-specific languages (DSLs) [**?**, **?**, **?**, **?**, **?**, **?**].

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

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

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

2.8 Acknowledgements

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

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

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

2.9 Conclusion

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

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

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

3 Data-centric Metaprogramming

3.1 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:

```
1 scala> (0, 1) * (0, 1)
2 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 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 (§3.4.5). Still, altering program semantics may be desirable—we exploit this property in the deforestation benchmark, shown in the evaluation section (§3.6).

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 [**?**]. 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 (§3.2);
- Presenting the extensions that allow global data representation transformations (§3.3) to be used as scoped programmer-driven transformations (§3.4);
- Implementing the approach presented as a Scala compiler plugin [**?**] that allows programmers to express custom transformations (§3.5) 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 [**?**]. These transformations produced speedups between 1.8 and 24.5x on user programs (§3.6).

3.2 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.

3.2.1 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 [**?**, **?**], 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:

```
1 def gcd(n1: (Int, Int), n2: (Int, Int)): (Int, Int) = {
2   val remainder = n1 % n2
3   if (remainder.norm == 0) n2 else gcd(n2, remainder)
4 }
```

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 §3.6):

```
1 scala> timed(() => gcd((544, 185), (131, 181)))
2 The operation takes 3.05 us (based on 10000 executions)
3 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:

```
1 scala> timed(() => gcdADRT((544, 185), (131, 181)))
2 The operation takes 0.23 us (based on 10000 executions)
3 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 (§3.6) 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.

3.2.2 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 §3.4, 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 §3.4.4.

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.

3.2.3 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 (§3.4.1)? How and when to encode and decode values, and what to do about values that are visible outside the scope (§3.4.2)? Even worse, what if parts of the code are compiled separately, in a different compiler run (§3.4.3)?

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 (§3.4.4).

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 (§3.4.5):

```
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.

3.3 Data Representation Transformations

As necessary background for our approach, we review data representation transformations and, in particular, the Late Data Layout transformation mechanism [**?**], 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.

3.3.1 Late Data Layout

The Late Data Layout (LDL) mechanism [**?**] is the underlying transformation used in Scala to implement multi-parameter value class inlining and to specialize classes using the miniboxed encoding [**?**]. 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`:

```
1 val i: Int = 1
2 val j: Int = i
3 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]

```
1 val i: int = 1
2 val j: int = i
3 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:

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

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.

[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 COERCE phase,

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

```
1 val i: @unboxed Int = 1 // expected/found: @unboxed
2 val j: @unboxed Int = i // expected/found: @unboxed
3 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:

```
1 val c: Boolean = ...
2 val l1: @unboxed Int = if (c) i else j
3 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 [**?**] 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 [**?**].

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:

```
1 val i: int = 1
2 val j: int = i
3 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.

3.3.2 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:

```
1 val a: @unboxed Int = 1
2 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:

```
1 val a: @unboxed Int = 1
2 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:

```
1 val a: @unboxed Int = 1
2 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 [**?**] 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.

3.4 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 3.2.2 showed how the ADR transformation is triggered by the `adrt` marker. The running example is reproduced below for quick reference:[2]

```
1 adrt(IntPairComplexToLongComplex) {
2   def gcd(n1: (Int, Int), n2: (Int, Int)): (Int, Int)={
3     val remainder = n1 % n2
4     if (remainder.norm == 0) n2 else gcd(n2, remainder)
5   }
6 }
```

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.

3.4.1 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:

[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 (§3.2) to mark the transformed version of the method.

```
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 (§3.6), 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 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 §3.3.2. Method call rewriting in `adrt` scopes is more general, and we describe it in section §3.4.4.

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 (§3.6) 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.

3.4.2 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 [**?**], 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:

```
1 adrt(IntPairToLong) { var x = (3, 5) }
2 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 §3.5):

```
1 var x: (Int, Int) = (3, 5) /* att: IntPairToLong */
2 val y: (Int, Int) = (2, 6) /* att: IntPairToDouble */
3 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 un-commenting 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 [**?**], 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.

3.4.3 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.

3.4.4 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 (§3.3.2):

```
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 $\star$, added via an implicit conversion (we will further analyze the interaction with implicit conversions in §3.4.5):

```
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.

3.4.5 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 §3.4.1.

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:

```
1 def use1(list: List[(Int, Int)]): Unit = ...
2 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]`:

```
1 adrt(ListOfIntPairAsListOfLong) {
2   def use3(list: List[(Int, Int)]): Unit = use1(list)
3 }
```

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

```
1 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 (§3.4.4).

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 §3.3, 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.

3.5 Implementation

We implemented the ADR transformation as a Scala compiler plugin [**?**], by extending the open-source multi-stage programming transformation provided with the LDL [**?**] artifact, available at [**?**]. 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:

```
1 def foo: (Int, Int) = {
2   /* id: 100 */ adrt(IntPairToLong) {}
3   /* id: 100 */ val n: (Int, Int) = (2, 4)
4   n
5 }
```

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:

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

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 §3.3 and §3.4.

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 §3.4.1. 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 [**?**]. 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.

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 3.1 – Greatest Common Divisor benchmark results.

3.6 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 [**?**], which warms up the Java Virtual Machine according to statistically rigorous performance evaluation guidelines [**?**].

3.6.1 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 contribution 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 [**?**].

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 §3.2;
- least squares benchmark + deforestation [**?**];
- 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 §3.2 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 3.1.

Benchmark	Time (ms)	Speedup	In-benchmark Garbage (MB)	In-benchmark GC time (ms)	Inter-benchmark Garbage (MB)	Inter-benchmark GC time (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 3.2 – Least Squares Method benchmark results.

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 [**?**, **?**] 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 §3.4.4. 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 [**?**], 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 3.2 shows 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: $(32 + 48 + 2 \times 16) * 5 \times 10^6 = 560 \times 10^6$, approximately 560MB of data.

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 3.3 – Sensor Readings and Hamming Numbers benchmark results.

The "blitz" transformation uses the dedicated collection optimization tool `scalablitz` [**?**, **?**] 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 [**?**] 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 [**?**], 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;
- 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 [**?**] and attempted to speed up:

```
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.

Benchmark	Generi	Minibo	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 3.4 – Scala Streams pipelines for 10M elements.

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.

3.6.2 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 [**?**, **?**], in order to replace standard library *functions* and *tuples* by custom, optimized versions adequate for miniboxed code [**?**]. The scope of this data representation transformation is miniboxing-transformed code.

The miniboxing transformation [**?**] 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 [**?**], which duplicates and adapts the generic code once for every primitive type, the miniboxing transformation only duplicates the code once

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 3.5 – Mapping a 1K vector.

and *encodes all primitive types in long integers*. This allows miniboxing to scale much better than specialization [**?**] 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`:

```
1 class $anon extends Function1[Int, Int] {
2   def apply(x: Int): Int = x + 1
3 }
4 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.

The Scala-Streams library

[**?**] 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 [**?**]. 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 3.4 shows the result of applying our data representation transformation to the Scala-Streams published benchmarks. (The benchmarks are described in detail in prior literature [**?**, **?**].) 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 [**?**, **?**]. This is a benchmark created by a third party (a commercial entity using Scala). Table 3.5 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.

3.7 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 [**?**] and class specialization [**?**, **?**]. 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 [**?**]. Rose has an analogous proposal for value classes [**?**, **?**] 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 [**?**] or inlining and escape analysis [**?**, **?**]. Furthermore, modern VMs, such as V8, Truffle [**?**] and PyPy [**?**] 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 [**?**] mechanism in the Scala setting. Similar approaches, with different specifics in the extent of type system and customization support, have been applied to Haskell [**?**]. Foundational work exists for ML, with Leroy [**?**] presenting a transformation for unboxing objects, with the help of the type system. Later work extends [**?**] and generalizes [**?**] such transformations. In terms of runtime-dispatched generics, we refer to the work on Napier88 [**?**] and the TIL compiler [**?**, **?**].

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

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

Multi-stage programming [**?**] is another technique that optimizes the data representation. Its Scala implementation, dubbed lightweight modular staging, can both optimize and even retarget parts of a program to GPUs [**?**, **?**]. 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 [**?**], based on Scala macros [**?**]. Another type-directed transformation in the Scala compiler is the pickling framework [**?**], 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.

3.8 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.

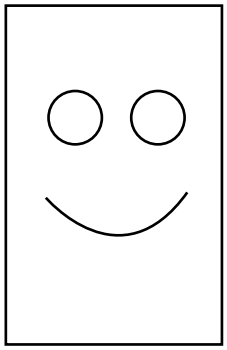
Acknowledgements

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.

A An appendix

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Personal details:



Name :	Mr. Sample CV
Address :	Samplestreet 70 6005 Luzern Switzerland
Date of Birth :	2nd of October 1981
Nationality :	Swiss
Legally work :	legally work in EU
Marital status :	with partner
Children :	none
Languages :	Chinese/Mandarin, English, French, German
Education level :	Bachelors degree
Hospitality work experience :	3-5 years
Special experience :	Europe work experience
Date of availability :	September 2009
Current location :	Africa
Travelling Status :	will be travelling single status
Telephone :	0041 41 370 6759
Email address :	jeff@h-g-r.com
Position(s) sought :	Permanent position for graduates
Department(s) sought :	Food & Beverage Bar/Sommelier

Personal profile:

As a Bachelor of Business Administration and after obtaining first relevant international work experience within the hospitality industry, I am now ready to take on new responsibilities to further my professional career.
My key strengths include strong analytical and logical skills, an eye for detail, communication and interpersonal skills.
I enjoy working in a team and help others progress. At the same time I work well independently.
As a highly motivated and driven individual I strive on taking up challenges.

Interests:

Travelling
Foreign Cultures
Photography
Sports

Educational qualifications:

Oct 99 - Feb 02	Higher Diploma (Hotel Management) Swiss Hotelmanagement School, SHL

Employment history:

Mar 04 - Ongoing	Assistant Manager (Rooms Division/Food & Beverage)
	Hotel Atlantic Kempinski Hamburg www.kempinski.com 5 star business hotel, part of Leading Hotels of the World 412 guest rooms, large function facilities, 3 food & beverage outlets
	Optimization of bar procedures, reinforcing SOPs
	Developing & implementing promotions
	Responsible for day-to-day operations
	Optimization and streamlining of housekeeping and laundry procedures
	Implementation of new SOPs
	Analyzing monthly reports for rooms division performance and sub departments
Mar 03 - Mar 04	Management Trainee
	Hospitality Graduate Recruitment www.h-g-r.com Leading company for placements within the Hospitality industry.
	Traineeship covering all aspects of an online recruitment agency.
Mar 02 - Mar 03	Management Trainee (Rooms Division)
	Hyatt Regency Xian, China www.hyatt.com 5 star business hotel 404 guest rooms, 4 food & beverage outlets
	Traineeship covering all rooms division departments on operational as well as supervisory level.

Training courses attended:

Mar 02 - Ongoing	OpenOffice - IT Courses
May 01 - Jan 03	Language Course - Chinese

References:

Hyatt Regency Xian
Patrick Sawiri, Phone: 86 22 2330 7654

Hospitality Graduate Recruitment
Jeff Ross, Phone: 41 41 370 99 88