Unifying Data Representation Transformations

Vlad Ureche   Eugene Burmako   Martin Odersky

EPFL, Switzerland

{first.last}@epfl.ch

Abstract

Values need to be represented differently when interacting with certain language features. For example, an integer needs to take an object-like representation when interacting with erased generics, although, for performance reasons, it normally uses the direct representation. In order to simplify the work of programmers, languages like ML and Scala expose the high-level concept (the integer) and let the compiler handle representation and conversion.

This pattern appears in multiple language features such as value classes, specialization and multi-stage programming mechanisms: they all expose a unified concept which they later refine into multiple representations. Yet, the implementations are typically ad-hoc and entangle the core mechanism with assumptions about the alternative representation and the implementation of generics, thus losing sight of the general principle.

In this paper we present an elegant and minimalistic type-driven generalization that subsumes and improves the state-of-the-art representation transformations. In doing so, we make two key observations: (1) annotated types conveniently capture the semantics of using alternative representations and (2) local type inference can be used to automatically, reliably and optimally introduce the necessary conversions.

We validated our approach by implementing three language features in the Scala compiler: value classes, specialization via miniboxing and a simplified multi-stage programming mechanism. An encouraging result is that we were able to reimplement and extend value class functionality in the Scala compiler with two man-weeks of work, without reusing any code from the previous implementation.

1. Introduction

Language and compiler designers are well aware of the intricacies of erased generics [12, 18, 25, 27, 29, 33, 36, 60], one of which is requiring object-based representations for primitive types. To illustrate this, let us analyze the `identity` method, parameterized on the argument type, `T`:

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

The low-level compiled code for `identity` needs to handle incoming arguments of different sizes and semantics: from bytes to long integers and references to heap-allocated objects. To implement this, compilers impose a uniform value representation, usually based on references to heap objects. This means that primitive types, when passed to generic methods need to be represented as objects, in a process called boxing. Since boxing slows down execution, whenever primitive types are used outside generic environments, they use their direct unboxed representation. In the example below, 'x' is using the unboxed representation denoted as `int`:

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

This example exposes two representations for the same concept: the high-level integer type `Int` can be represented either as an unboxed primitive `int` or as a boxed `Object`, which is compatible with erased generics. There are two approaches to implementing this duality: In Java, both the boxed and unboxed integers are accessible, thus making programmers responsible for choosing the representation and converting when necessary. To avoid burdening the programmers with implementation details, the ML and Scala programming languages expose a unified concept of integer and automatically choose the representation inserting the necessary conversions during compilation.

This mechanism of exposing a single concept with multiple representations is used in other language features as well:

Value classes [1, 5, 24] behave as classes in the object-oriented hierarchy, but are optimized to efficient C-like structures [55] 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 [18, 19] 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 [4, 60], 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 (referred to as "staging") [57] allows executing a program in multiple stages, at each execution stage generating a new program that is compiled and ran, until the final program outputs the result. In practice, this technique is used to lift expressions to operation graphs and 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.

The examples above seem like unrelated language features and mechanisms. And, indeed, compiler designers have provided dedicated solutions for each of them. These solutions are typically designed in an ad-hoc way, addressing just the problem at hand. For instance, the solutions employed by ML and Scala are aimed at satisfying the constraints of erased generics [12, 33, 59], and hardcode this decision into the transformation algorithm. Miniboxing uses a custom transformation implemented as a Scala compiler plugin [4, 60]. The Lightweight Modular Staging framework [44] in Scala is implemented using a fork of the main compiler, dubbed Scala-Virtualized [35] which is specifically retrofitted to support staging.

Yet, all these mechanisms have two elements in common: (1) the use of multiple representations for the same concept and (2) the introduction of conversions between these representations. These two similarities suggest that there is an underlying yet undiscovered principle that generalizes the individual transformations. We believe exposing this principle will disentangle the transformations from their assumptions, enabling researchers to reason about them and implementors to reuse a common infrastructure.

To this end, we present an elegant and minimalistic type-driven mechanism that uses annotations to guide the introduction of conversions between alternative representations. In doing so, we make two key observations: (1) annotated types conveniently capture the semantics of using an alternative representation and (2) the type system can be used to automatically, reliably and optimally introduce conversions between the representations.

Annotated types are a mechanism that allows attaching additional metadata to the types in a program [3, 6]. This in turn allows external plugins to verify additional properties of the code while leveraging the existing type system. Indeed, annotations have been used to statically check a range of program properties, from simple non-`null`-ness to full effect tracking [40, 49].

Our first key insight is that annotated types are a perfect match for the one-concept-multiple-representations approach. The semantics of annotated types can be specified externally and can change during the compilation pipeline, so we can emulate the concept by keeping annotated and non-annotated types compatible before introducing conversions and later we can emulate the two representations by making the types incompatible. Furthermore, unlike other mechanisms, annotated types allow us to selectively mark the values that will use 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 its boxed representation.

Depending on the transformation, annotations can be introduced automatically by the compiler or manually by programmers. This provides the flexibility necessary to capture a wide variety of transformations: some of the transformations work automatically, e.g. unboxing primitive types and value classes, whereas others, like staging, where annotations represent domain-specific knowledge, require manual annotation.

This flexibility of annotating values that use the alternative representation is in sharp contrast to state of the art transformations for erased generics [12, 33]. These transformations consider the unboxed representation as always desirable and hardcode the semantics of erased generics into their transformation rules. 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 or reified and unboxed [7, 29]. This flexibility is fundamental to staging, but also enables a better translation in the case of erased generics, as the next sections will explain:

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

The type system can be used to automatically, reliably and optimally introduce conversions based on the annotated types. Since the semantics of annotated types can change during compilation, we can trigger the separation by making annotated types incompatible with non-annotated type. Then, retypechecking the abstract syntax tree will expose incompatible types, which correspond to representation mismatches that we need to patch by introducing conversions.

Our second key observation is that name resolution and type checking for annotated types provides what can be seen as a forward data flow analysis [30] for representations. On the other hand, local type inference [38, 42] propagates expected types, and thus provides a backward data flow analysis. Having these two analyses meet at points where the representation doesn't match ensures that conversions are introduced only when necessary:

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

Being type-driven, our approach can be seen as a generalization of the work of *Leroy* on unboxing primitive types in ML [33]. Yet, it is far from trivial generalization: (1) we introduce the notion of selectively choosing values to use the alternative data representation, which is crucial to enabling staging and bridge methods [16] in object-oriented hierarchies, (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 unboxed primitive types.

Domain-specificity of particular data representation transformations can be expressed with custom logic that makes use of annotations and conversions obtained through the type system. For example, primitive unboxing will replace `@unboxed Int` by `int` and give semantics to `box` and `unbox`, in this case creating the boxed object and accessing its `value` field respectively.

The paper makes the following contributions:

- We survey existing approaches to data representation transformations (§2 and §3), and show there is a need for a novel approach that allows for additional flexibility;
- We show a general data representation transformation mechanism for the annotated program, which does not impose the semantics of the alternative representations nor of the conversions (§4) and make the first steps towards formalizing it (§5);
- We validate the mechanism by implementing three language features in Scala compiler using our transformation: value classes[1], specialization using the miniboxing representation[2] and a simple staging mechanism[3] (§6).

In the following sections we provide detailed motivation, elaborate on the details of the mechanism and validate our approach.

2. General Data Representation

In this section we present several data representation transformations that deal with boxing and unboxing of primitive types, highlighting their strong and weak points on small examples. We start with a naive approach, continue with a transformation that eagerly introduces conversions and conclude with an on-demand transformation, which aims at introduces conversions only when necessary.

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 convert it to the unboxed integer, `int`. We follow the same convention for other primitive types, such as `Long` and `long`, `Byte` and `byte` or `Double` and `double`. Furthermore, 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 before types (e.g. `@unboxed Int`) instead of after (e.g. `Int @unboxed`), as the Scala syntax requires.

2.1 Naive Transformations

In order 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 1-element linked list with this element:

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

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

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

The resulting code is invalid. In the first statement, `x` is unboxed while the right-hand side of its definition, the head of a 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.

2.2 Eager (or 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, converting to the right representation:

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

In the snippet above, two conversions 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 conversions we can keep the program code consistent. Let us take another example:

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

Translating this code using the eager transformation produces the following output:

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

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

Since a is transformed from boxed to unboxed, all its occurrences are replaced by box(a). The same happens for b, so its definition is wrapped in an unbox call. Yet, this produces suboptimal code, which boxes a just to unbox it later. In the case of Scala programs, which are compiled to Java Virtual Machine (JVM) bytecode, this redundancy can be eliminated by escape analysis [54] in the just-in-time compiler [31, 39]. Yet it typically takes 10000 executions to trigger the just-in-time compiler and the optimizations, which means 10000 boxed integers are created just to be immediately unboxed and garbage collected later.

2.3 Peephole Optimization For Eager Transformations

In order to cut down redundant conversions in code produced by eager transformations, we need to perform a second pass over their result using a technique called peephole optimization. Our optimization rewrite box(unbox(t)) and unbox(box(t)) to just t. This simple transformation, similar to the ones in Haskell [27] and miniboxing [60], can be successfully used to eliminate the redundant conversions in the definition of b.

Yet, removing just the aforementioned conversions is not enough. The following example shows this:

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

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

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

Therefore, we need an extra rule for distributing the unboxing operation inside: unbox(t1 + t2) ⇒ unbox(t1) + unbox(t2).

In practice we have seen many more examples that require additional optimization rules. This suggests that although eager transformations work well for minimalistic intermediate languages, such as Haskell's Core, the complexity of accompanying peephole optimizations make them impractical for Scala. The initial implementation of miniboxing [60] used an eager transformation, but we were forced to look for alternative approaches due to the number and complexity of the peephole optimization rules.

2.4 On-Demand (or Type-driven) Transformations

As outlined above, eager transformations introduce representation conversions at all definition and use sites of primitive values. This makes them straightforward, but also suboptimal, producing extraneous conversions that later need to be cleaned up.

An alternative approach would be to introduce conversions only when a representation mismatch occurs, using a dedicated mechanism to check representation consistency. For example, making the boxed/unboxed distinction explicit to the type checker allows its logic to check whether representations correspond and hence whether conversions need to be introduced. This achieves optimality in the case of a + b, because the type checker will know that all variables are unboxed, and hence no conversions are necessary:

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

This idea is a precursor to the transformation presented in this paper, but there are still issues that need to be addressed before we get to our final unified approach.

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, these cases are common: several value classes can have the same parameter types, so their unboxed representations coincide and all staged expressions share the same alternative representation. With this in mind, let us consider the following piece of source code:

```
val m: UInt = 1
val n: Int = -1
List(if (...) m else n)
```

On the one hand, both m and n are unboxed, which means that the corresponding if expression is also unboxed to int. On the other hand, the constructor of the linked list expects a boxed argument. This suggests that the if expression needs to be boxed, but actually in this situation boxing is impossible, because int, the type of the if, can box both to Int and UInt, and we can't discern between the two based on the context. The correct translation should have been:

```
val m: int = 1
val n: int = -1
List(if (...) box[UInt](m) else box[Int](n))
```

It may seem that transforming values one by one might provide a way out of the conundrum. This way, only a single value at a time would be in flux, which would make the choice of boxed representation unambiguous.

However, this takes us back to the square one with respect to (non)-optimality of the resulting code, because in an if expression both branches to be transformed at once - either both becoming boxed or both unboxed, and that can't be decided in an optimal way based on the type of just a single branch. For example, while boxing by default is desirable in the example above, it would not be desirable if both m and n were signed integers, in which case it would be best to box the entire if expression back to Int instead of the individual branches.

Clearly, a different perspective is required to make on-demand transformations viable, and this is something that we'll present in subsequent sections.

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 for

Scala. This section identifies additional challenges introduced by object orientation.

3.1 Subtyping

In languages like Java and C#, all types have a common supertype, usually called `Object`, which provides universal methods such as `toString`, `hashCode` or `equals`. This presents a challenge for unboxing primitive types:

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

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

3.2 Virtual Method Calls

Virtual method calls also create challenges for data representation transformations. Boxed objects can act as the receivers of virtual calls, because they have an associated virtual method table. However, unboxed objects only store the contents of their fields and nothing more, which complicates matters:

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

There are two approaches to handling virtual calls: 1) an unboxed callee can be boxed so it can act as a receiver of the virtual call, or 2) if the corresponding method is final, then its implementation can be extracted into a static helper, making the virtual method table unnecessary. 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 [5]:

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

3.3 Necessity for Selective Transformations

We argue that selectivity should be built into data representation transformations as a first-class concern allowing the compiler or the programmer to individually pick the values that will use the alternative representation. At the moment, most data representation transformations make the assumption that all values that can use an alternative representation should also use it. However we identified several cases where this assumption is invalid:

The low level target language may impose certain restrictions on the representations used. For example, the Scala compiler targets JVM bytecode, which 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 transformation is selective;

Bridge methods [16] are introduced to maintain coherent inheritance and overriding relations between generic classes in the presence of erasure and other representation transformations. Bridge methods are introduced when the low level signature of a method does not conform to one of the base method it overrides. Consider the following example:

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

A naive translation of this code that doesn't account for erasure will end up with `F.id` having the low-level signature `(d: int): Unit`, which on the JVM 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 unboxed argument. Such method will be correctly perceived as overriding `E.id` by the JVM, and then it can dispatch to the unboxed implementation:

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

Generating this code is impossible if the data representation transformation always unboxes `D`, which is again only possible if the transformation is selective.

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

```
def reverse(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
}
```

Unless the data representation transformation is selective, in the low-level bytecode, `elt` is represented as an unboxed value. But during each iteration, assigning the `head` of the (generic) list to `elt` converts a boxed integer to the unboxed representation. The subsequent statement performs the inverse transformation, creating a new boxed integer from `elt` and prepending it to the reversed list. This sequence of conversions in the hot loop severely impacts the performance, making it desirable to selectively enable unboxing.

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

4. Unified Representation Transformation

This section presents our unified data representation transformation mechanism. We start by explaining the key insights of the transformation, then dive into the algorithm itself and finally show how this transformation improves over the state of the art.

4.1 Key Insights

In a compiler, name resolution is effectively the high-level equivalent of a forward data flow analysis [17], tracking the reaching definitions. Coupled with a type system, name resolution propagates the types of symbols in a program's syntax tree. On the other hand, the local type inference [38, 42] mechanism in a type checker acts as a backward data flow analysis tracking the expected type of expressions.

The first key insight is that name resolution and local type inference can effectively collaborate to produce a data flow analysis which can find mismatching representations and trigger conversions between them:

```
def foo(x: String): Int =
  x // forward analysis: x refers to argument
    // x of method foo of type ‘String‘
    // backward analysis: the return type of
    // method foo needs to be ‘Int‘
    //      => types do not match
```

Forward and backward tracking of type and thus representation information allows the optimal transformation of code, as shown in the lazy transformation description in §2.4. Optimality comes from introducing conversions only in the case of a mismatch, and therefore not introducing redundant conversions (excluding situations with explicit domain-specific decisions to not unbox certain values that are better off boxed, as shown in §3.3).

Use of types to encode data representation is widely spread, having been employed in a number of data representation transformations [5, 27, 33]. Yet unboxing directly to the primitive type will not work unless the mapping is reversible (§2.4). A different approach is to introduce new synthetic types [5] for the alternative representation, but this is inflexible: (1) the types are not accessible to programmers, thus the programmers cannot intervene in the data representation decisions and (2) such types are typically designed for the problem at hand and do not generalize. From a bird's eye view, the problem with new synthetic types is that they only model the unboxed representation, whereas the ideal type encoding would allow modelling concepts that are later split into different representations.

In this context, the second key insight regards tracking representation information in the type system: annotated types [3, 6] are a perfect fit for tracking unboxed representations: (1) they are accessible to programmers, (2) the mapping from boxed to unboxed types is reversible and (3) they can change semantics during compilation. For instance, for the compiler phases before the data representation transformation, `@unboxed Int` and `Int` can be fully compatible, while afterwards they can become incompatible, guiding on-demand transformations into introducing conversions.

4.2 Mechanism

Having seen the key insights, we can now dive deeper into the generic transformation mechanism. It is composed of three phases: *inject*, *convert* and *commit* that work with type-checked programs. Throughout the presentation we will use the following example:

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

The inject phase just marks the types that are going to be unboxed without doing anything else. In particular, no conversions are performed during this phase, which allows marking values, changing method signatures and redirecting method calls without worrying about representation incompatibilities or introducing conversions. In the example below, this is most visible in the recursive call to fact, where the argument is not converted to a different representation:

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

Unlike in previous examples, now we explicitly mark the constant literals for unboxing: the literal constant `1` will be an unboxed value in the low level code, thus we mark it for unboxing using `1: @unboxed`. Also, the operators in Scala are desugared to method calls, and we made this explicit by adding the commonly accepted method notation: `receiver.method(args)`. Thus, instead of `n <= 1` we wrote `n.<=(1)`. These expansions do not impact the generality of the transformation but serve to clarify how the transformation mechanism works.

This phase can have other uses as well, such as, for example, introducing bridge methods or duplicating code and specializing it, in the case of miniboxing. Alternatively, it might not be present at all, as in the case of staging, where the values to be lifted are annotated by the programmer.

The convert phase is the centerpiece of the algorithm and is similar for all data representation transformations. It has two steps: (1) it makes the annotated types incompatible with the un-annotated types, thus invalidating the current abstract syntax tree and (2) it re-typechecks the tree and introduces conversions where necessary. The conversions are introduced when an annotated type is expected and the un-annotated type is passed and vice-versa. Since the tree has already passed name resolution and typechecking before, running the typechecking algorithm again will only make a difference when conversions need to be inserted.

Additionally, note that the object-oriented aspects of the language need to be taken into account. For example, method calls on the alternative representation require boxing, which can later be removed by the commit phase using extension methods [5]. Supertypes are also taken into account by this phase. Since types with alternative representations are final, a class cannot have an alternative representation and be a superclass at the same time. Therefore, supertypes will not be annotated. If an unboxed value is assigned to its supertype, a boxing conversion is automatically inserted due to the annotation mismatch. With the conversion in place, we have a boxed value that is indeed compatible with the supertype.

The convert phase will transform our running example to:

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

In this listing, `box` conversions are introduced for unboxed method call receivers while the `unbox` conversions are used to convert the boxed results of the `*` and `-` operators to unboxed integers. The resulting code has boxing and unboxing semantics of the program expressed in a clear, explicit form that is used by the final phase, commit.

The commit phase is the final phase in the transformation mechanism and is meant to give semantics to the annotated types and to the conversions. For instance, in a primitive type unboxing transformation, commit is going to transform `@unbox Int` into `int`, `unbox` into a field access that extracts the unboxed value from a boxed integer, and `box` into an object creation. Method calls can also be redirected to extension methods (in this case underlying platform's intrinsics), thus cutting down on object allocations. After the commit phase the program is fully and optimally transformed:

```
def fact(n: int): int =
  if (intrinsic_<=(n, 1))
    1
  else
    intrinsic_*(n, fact(intrinsic_-(n, 1)))
```

4.3 Advantages

The previous sections have shown the motivation for the transformation and how our mechanism works. We will now elaborate on how our work improves upon the state of the art in data representation transformations.

Optimality. Unlike eager transformations in use today [27], our mechanism tracks representations in a forward and backward data flow analysis to avoid introducing redundant conversions. This makes tedious peephole transformations redundant and enables the data representation transformation to focus on the most important aspect: correctly transforming the program in the inject and commit phases.

Selectivity. Selectivity is build into the transformation mechanism as a first-class concern. Indeed, in the previous example, our choice was to transform both the input argument of the `fact` method and its return type. But selectively transforming one or the other is equally simple: by not annotating the argument or the return value, one can derive different scenarios of the transformation, all with the exact same convert and commit transformations.

Generality. Using the proposed transformation mechanism eliminates the need for peephole optimizations (§2.3) and drastically simplifies the overall specification: only the rules for the inject and commit phases need to be specified, since the convert phase is reused from one transformation to the next. In our validation section (§6), we describe three scenarios where we used the exact same convert mechanism, and only needed to reimplement the inject and commit transformations. This shows the mechanism is general and makes transformation specification simple and elegant. Furthermore, the transformation mechanism is flexible in that it can accommodate a wide range of data representation transformations.

Support for **object-oriented aspects** in a language is provided by the simple boxing of method call receivers and expressions for which a supertype is expected. Furthermore, the commit phase can reduce the boxing operations by replacing virtual method calls by static extension method calls, thus reducing the number of heap objects allocated.

The proposed mechanism provides a strong contribution in an unexpected direction, considering its ingredients: local type inference and annotated types. We have validated this mechanism in three scenarios with very different requirements and representations. The next section will provide a formal description of the convert phase.

5. Formalization

This section sets out the formal rules of the transformation. Despite presenting the rules, we do not prove operational equivalence in this paper and leave it for future work.

The formalization for the convert transformation is based on System $F_{<:}$ [41] with local colored type inference [38, 42]. We use the notation ${}^{\vee}T$ for types synthesized from terms and ${}_{\wedge}T$ for inherited (expected) types. Intuitively these two notations correspond to the forward and backward propagation in a data flow analysis.

To support annotated types, we extend the types of System $F_{<:}$ to:

$$\begin{array}{llr}
T \quad ::= & & \text{types:} \\
 & X & \text{type variable} \\
 & Top & \text{maximum type} \\
 & T \rightarrow T & \text{type of functions} \\
 & \forall\, X <: T.\, T & \text{universal type} \\
 & \texttt{@alt}\; T & \text{annotated type}
\end{array}$$

5.1 Before the Convert Phase

The input for the inject phase is a typed tree, with all the types inferred and the expansions (such as implicit parameters) in place. In all the phases before the convert phase we have the following two subtyping rules:

$$T \;<:\; \texttt{@alt}\; T \qquad \text{(S-ALT1)}$$

$$\texttt{@alt}\, T \ <:\ T \qquad \text{(S-ALT2)}$$

These two rules ensure that annotated and non-annotated types are compatible, thus modelling the unified concept.

5.2 After The Convert Phase

Passing to the convert phase, the S-ALT1 and S-ALT2 subtyping rules are removed. We then retypecheck the program with two additional rules for annotated types. We use the following notation for the typing judgement:

$$T, \Gamma \vdash t \rightsquigarrow t' : T'$$

Where T is the expected type, Γ is the typing context, t is the original term, which is rewritten to t' and T' is the actual type. The two additional rules introduce conversions when representations do not match:

$$\frac{{}_\wedge\texttt{@alt}\ {}_\wedge T, \Gamma \vdash t \rightsquigarrow t' : {}^\vee T' \qquad {}^\vee T' <: {}_\wedge T}{{}_\wedge\texttt{@alt}\ {}_\wedge T, \Gamma \vdash t \rightsquigarrow \texttt{unbox}\, t' : {}_\wedge\texttt{@alt}\ {}^\vee T'}\ \text{(T-ALT)}$$

$$\frac{{}_\wedge T, \Gamma \vdash t \rightsquigarrow t' : \texttt{@alt}\ {}^\vee T' \qquad {}^\vee T' <: {}_\wedge T}{{}_\wedge T, \Gamma \vdash t \rightsquigarrow \texttt{box}\, t' : {}^\vee T'}\ \text{(T-UNALT)}$$

The intuition behind the first rule is that whenever the expected type is annotated, but the term's type is not, the rule introduces an explicit `unbox` operation. And vice versa, the second rule will trigger when a term's type is annotated but the expected type in not, thus the `box` operation is introduced. The `box` and `unbox` functions are considered built-in, with the expected generic signatures. It is worth noting that the typing rules will only introduce the conversions to terminal AST nodes, namely to those in which representations happen to mismatch:

```
def fact(n: @unboxed Int, l: List[Int]): Int =
 if (...)
   l.head
 else
   box(n) // the expected type Int propagates
          // to the then and else branches, of
          // which the else branch needs boxing
```

6. Validation

This section describes how we validated the unified transformation mechanism by using it to implement three very different language features: value classes, specialization via miniboxing and staging.

In our case studies we observed increased productivity thanks to the reuse of the common transformation mechanism in all three scenarios. Two decisions also provided tangible benefits to the development process: (1) decoupling the decision to unbox values from the actual mechanism for introducing conversions and (2) decoupling the alternative representation semantics from the conversions and annotated types. A highlight of the validation is the fact that we reimplemented and extended the Scala compiler support for value classes [5] with just two man-weeks of work and without reusing any pre-existing code.

We will begin by describing the plugin architecture in the Scala compiler and how it can be used to implement data representation transformation, and continue by presenting each of the three case studies.

6.1 Scala Compiler Plugins

The Scala compiler allows extension via compiler plugins. These can customize the type checker via analyzer and macro extensions and can also add new compilation phases. In this section we will describe the annotation checker framework that is part of analyzer extensions and present the custom compiler phases in data representation transformations.

With the annotation checker framework, compiler plugins can inject annotations during typechecking, can provide custom logic to calculate meets and joins of annotated types, and can apply custom transformations to trees that have annotated types. Moreover, in this framework it is also possible to extend the vanilla subtyping logic in the Scala compiler by providing custom and phase-dependent subtyping rules for annotated types.

The annotation checker framework can therefore attach custom semantics to annotated Scala types. Using this framework, *Rytz* created a purity and effects checker [49] that uses annotations to track side-effecting code, and *Rompf* implemented a continuation-passing style (CPS) transformation triggered using annotated types [45].

In the context of data representation transformations, the **annotation checkers** framework is used to implement the one-concept-multiple-representations mechanism. Before the convert phase, the annotated types are compatible with their non-annotated counterparts, therefore allowing the transformations to rewrite the code without taking representations into account. During the convert phase, the annotated types and non-annotated types become incompatible, driving the insertion of `box` and `unbox` conversions. Finally, after the convert phase, the types remain incompatible, so further phases do not confuse representations:

```
def annotationsConform(tpe1: Type, tpe2: Type) =
 if (phase.id < convertPhase.id)
   true
 else
   (tpe1.isAnnotated == tpe2.isAnnotated) ||
      tpe2.isWildcard
```

In order to transform code, compiler plugins can also introduce **custom phases**, at precise points in the compiler pipeline. A data representation transformation plugin typically creates three custom phases, corresponding to the inject, convert and commit phases in the mechanism outlined in §4.

The inject phase is used to inject annotations that initiate the transformation process. To do so, the phase visits all entries in the symbol table and updates their signatures by annotating types that need to be converted to alternative representations. Since this phase is dependent on the transformation at hand and typically does more than just adding annotations, it will be described in more detail in each of the case studies.

The convert phase is the core of the transformation mechanism and is similar for all case studies. Since the inject phase changes the symbols' signatures and the annotation checker makes the annotated types incompatible, the convert phase essentially starts with an inconsistent abstract syntax tree, in the sense that, in its initial shape, it does not typecheck. The job of the convert phase is to introduce conversions where necessary such that tree is consistent again. This is done by re-typechecking the tree and introducing conversions when representation mismatches occur.

Following the framework described in §4 and §5, the convert phase retypechecks the program, making use of the local type inference algorithm implemented in Scala in order to perform backward and forward propagation of representation information.

In Scala, the type checker consists in two parts: (1) the typing judgement procedure, which can assign types based on the typing context and (2) the adaptation routine, which applies fixups to trees such that their types match expected types. A typical use for the adaptation is inserting implicit conversions, resolving implicit parameters and synthesizing reified types [51].

The convert phase overrides the typing judgement with a single extra rule: the receivers of method calls must be boxed. This, along with boxing when supertypes are expected, which is done automatically, forms our support for object orientation. We also override the adaptation routine, which allows us to implement the T-ALT and T-UNALT rules for introducing conversions.

Finally **the commit phase** runs and updates the symbol signatures and the conversions to account for the exact semantics of the alternative representation. Thanks to the previous phase, the tree is consistent and conversions act as markers that the commit phase can use to transform the tree. Again, since this phase is specific to react transformation, we will describe it in each case study.

6.2 Case Study 1: Value Classes

Value classes [1, 5, 24] marry the homogeneity and dynamic dispatch of classes with the memory efficiency and speed of C-like structures. In order to get the best of both worlds, value classes have two different in-memory representations. Instances of value classes (referred to as value objects) can be represented as fully-fledged heap objects (the boxed representation) or, when possible, use a struct-like unboxed representation with by-value semantics.

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

```
@value class Meter(val x: Double) {
 def +(other: Meter) = new Meter(x + other.x)
 def <=(other: Meter) = x <= other.x
}
def report(m: Meter) = {
 if (m.<=(new Meter(9000))) println(m.toString)
 else println("it's over nine thousand")
}
```

Before we dive into the transformation, let us consider some basic facts about value classes, correlating them with existing implementations for C# [1] and Scala [5] (both the official transformation shipped with Scala 2.10+, and the prototype we present in this paper).

Finality. Even though value classes can be perceived as classes, their participation in subclassing 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 extends `Meter`, then boxing `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 a marker type `AnyVal` and are automatically made final by the compiler.

By-value semantics. When compiling value classes down to low level, additional care must be taken to accommodate their by-value semantics on otherwise object-oriented platforms. For instance, both the JVM and the CLR have a universal superclass called `Object` that exposes by-reference equality and hashing. Moreover, both languages 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 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 into their single field, 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 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, convert and commit.

The inject phase transforms signatures of all fields, locals and parameters of value class types as well as return types of methods that produce single-field value objects by annotating them with `@unboxed`:

```
@value class Meter(val x: Double) {
  def +(other: @unboxed Meter): @unboxed Meter =
    new Meter(x + other.x)
  def <=(other: @unboxed Meter) = x <= other.x
}
def report(m: @unboxed Meter) = {
  if (m.<=(new Meter(9000))) println(m.toString)
  else println("it’s over nine thousand")
}
```

This is a notable use-case for the first-class selectivity support provided by our 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. Leaving off the `@unboxed` annotation is all that it takes to have the result automatically boxed in the method and unboxed at the caller.

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

```
trait Reporter[T] {
  def report(x: T): Unit
}
class Example extend 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 boxed. This will produce a compatible signature for the JVM and the convert phase will introduce the correct conversions.

The convert phase follows the pattern established in §4, making `@unboxed` types incompatible with their non-annotated counterparts and inserting `box` and `unbox` markers in case of representation mismatches. For our running example, the following code will be produced:

```
@value class Meter(val x: Double) {
  def +(other: @unboxed Meter): @unboxed Meter =
    unbox(new Meter(x + box(other).x))
  def <=(other: @unboxed Meter) =
    x <= box(other).x
}
def report(m: @unboxed Meter) = {
  if (box(m).<=(unbox(new Meter(9000))))
    println(box(m).toString)
  else println("it’s over nine thousand")
}
```

The commit phase uses the annotations established by the inject phase and the marker conversions inserted by convert in order 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, duplicating the declarations as necessary for 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 conversion 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` field selections to references to `e` or one of its duplicates and transforming non-virtual `box(e).m(args)` method calls into calls to static extension methods.

Finally, term transformations perform necessary bookkeeping to account for duplicated declarations (arguments to 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).

```
final class Meter(val x: Double) {
  def +(other: Double) = Meter.+(x, other)
  def <=(other: Double) = Meter.<=(x, other)
}
object Meter {
  def +(x: Double, other: Double) = x + other
  def <=(x: Double, other: Double) = x < other
}
def report(m: Double) = {
  if (Meter.<=(m, 9000))
    println(new Meter(m).toString)
  else
    println("it’s over nine thousand")
}
```

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, convert and commit phases spans only about 250 lines of code. This shows that our mechanism can significantly reduce the effort necessary to implement complex data representation transformations.

6.3 Case Study 2: Miniboxing

The miniboxing transformation in Scala [4, 60] is the most complex transformation of the three case studies, and it is also the most established, being under development for almost two years. This section briefly mentions the ideas behind miniboxing and goes on to present how the data representation mechanism was used in the miniboxing plugin.

Specialization [18] improves the performance of erased generics: aside from the generic class, specialization creates adapted variants for each primitive type. The variant classes offer specialized methods, which receive and return primitive types, therefore allowing the program to use the class without boxing primitive types. Yet, specialization leads to bytecode duplication, with 10 classes per type parameter: 9 for the primitive types in Scala plus the erased generic class. This means that specializing a tuple of 3 elements, which has 3 type parameters, produces 10^3 classes.

Miniboxing was designed to reduce the bytecode explosion in specialization. There are two key insights: (1) in Scala, all primitive types can fit into a single tagged union containing a type tag and a long integer payload, thus reducing the duplication to two classes per type parameter and (2) since Scala is strongly typed and all primitive types are final, all values of type `T` are statically guaranteed to have the same tag, which means we can attach tags to code instead of values – effectively hoisting the tag and obtaining a lightweight reified types scheme [51], where a single tag is passed for the type parameter. With miniboxing, fully specializing a 3-element tuple creates 8 classes and an interface.

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

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

The `@miniboxed` annotation on type parameter `T` triggers the miniboxing transformation of the method. This will duplicate and adapt the body of `identity` to create `identity_M`, which accepts primitives. This new method encodes the primitive types in Scala into a long integer and additionally receives a hoisted type tag corresponding to the reified type of `T`. The low level code will be:

```
def identity(t: Object): Object = t
def identity_M(tag: byte, t: long): long = t
minibox2int(identity_M(INT, int2minibox(5)))
```

In getting to this low level code, the **inject phase** duplicates the method `identity` to `identity_M` and adds the type tag:

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

In the new method, the miniboxing plugin needs to transform all values of type `T` to `Long`. `Long` corresponds to the payload in the tagged union, and is capable of storing the value of any primitive type in Scala. The initial version presented in [60] used an eager transformation coupled with a peephole optimization, but the number and complexity of the peephole rules made this unfeasible. This motivated the development of the data representation mechanism. Using this mechanism, the inject phase marks the values that will use an alternative representation (miniboxing uses `@storage` for the annotated types):

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

Furthermore, the inject phase has the additional role of redirecting calls from the generic versions of the method to the miniboxed variants whenever the type arguments are instantiated with primitive types or are known to be miniboxed type parameters of an enclosing class or method:

```
identity_M(INT, 5)
```

Going into the **convert phase** the two methods do not change, but the call to `identity_M` needs conversions:

```
unbox(identity_M(INT, box(5)))
```

The `box` and `unbox` methods serve as markers, that the **commit phase** replaces by `minibox2box`, or, if the target type is known, to more specific conversions such as `minibox2int` (and vice-versa). During the commit phase, annotated types are also converted to the `Long` integer, which is the alternative representation in miniboxing:

```
def identity[T](t: T): T = t
def identity_M[T](tag: Byte, t: Long): Long = t
minibox2int(identity_M(INT, int2minibox(5)))
```

Finally, as this code passes through the Scala compiler's backend, the primitive unboxing and erasure phase transforms boxed `Long` integers into unboxed `long` and erases the type parameter `T` to `Object`. This produces the exact result we showed earlier.

It is worth mentioning that miniboxing exploits all the flexibility available in the data representation mechanism: the alternative representation mapping is not injective, since all miniboxed type parameters map to `Long`, the selectivity is used to generate bridge methods for similar reasons to those presented in §3.3 and the compatibility between annotated and non-annotated types in the inject phase is used to easily redirect method calls.

The miniboxing plugin [4] is now accepted in the Scala community and several projects are experimenting with it. This shows that the unified data representation mechanism is not just a prototype but can reliably transform large code bases.

6.4 Case Study 3: Staging

Multi stage programming [57] 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 [44, 46], which removes the cost of abstractions in many high-level embedded DSLs [43]. Yet, using LMS is not straightforward, as it requires a custom version of the compiler, dubbed scala-virtualized [35], which is capable of lifting built-in language constructs. In this section, we will show how this can be done as a data representation transformation.

One of the early examples of staging given by *Rompf* is partially evaluating a power function through staging:

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

The `pow` method computes b^e. The base, `b`, is marked as lifted, whereas the exponent, `e`, is not. This means that calls to `pow`, instead of computing a value, accumulate the operations necessary to reproduce the result for a (possibly unknown) argument and a fixed exponent `e`. The call to `function` triggers the execution of `pow` for a stage-time argument (`arg`) and fixed value of the exponent, in this case 5. This is followed by outputting the recorded operations and re-compiling them into more efficient code. These recorded operations do not include `if` statements, since `e` is a stage-time constant and unroll the recursive call to `pow`:

```
Compiling the following code:
**********************************
(arg0: Double) => {
  val x0: Double = arg0 * arg0
  val x1: Double = x0 * x0
  val x2: Double = arg0 * x1
  x2: Double
}
**********************************
3.0^5 = 243.0
4.0^5 = 1024.0
```

The key to staging `pow` is that the `@lifted` arguments are replaced by an alternative representation that accumulates expressions into an operation graph and can synthesize new code for them. In this case, the **inject phase** doesn't exist at all, since the user marks the arguments to be `@lifted` in the source code.

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

The **commit phase** is the most interesting in the transformation: it redirects method calls from the alternative representation to a special staging object that records the operation graph. This operation graph is then used to generate optimized code. It also redirects the `function` call from the identity (in case the program is not staged) to `function_compile`, which triggers the synthesis of the code for the final result and compiles it. 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* [43, 44, 46] for more details.

The staging prototype we implemented serves to show that lifting is yet another case of using an alternative representation. Although the simple example we have shown could also be staged with the normal Scala compiler, for more complex examples we argue that instead of using a custom Scala compiler [35], lifting should be treated as a representation transformation, which would allow maximizing infrastructure reuse and making staging more accessible to developers.

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 [18], compile-time C++ template expansion [55] and load-time template instantiation [29] as done by the CLR [7]. Homogeneous translations, on the other hand, require a uniform data representation, which may be either boxed values [12, 33], fixnums [62] or tagged unions [36].

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 [51], several solutions have been proposed to reduce it: in the CLR, reified types are computed lazily [29]. In Java several papers have been published explaining schemes for carrying reified types, including PolyJ [8], Pizza [37], NextGen [16] and the work by *Viroli et al.* [61]. Finally, in ML generic code (also called parametrically polymorphic in functional languages) carries explicit type representations [25, 58].

Unboxed primitive types. In the area of unboxed primitive types, *Leroy* [33] presents a formal data representation transformation for the ML programming language based on typing derivations. The comparison in the introduction states that we introduce selectivity, object-oriented support and disentangle 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 hardcoded while the two versions of the transformation presented correspond to the typing algorithm in the convert phase for the case of annotated and non-annotated expected type. Instead of expected types, the transformation knows where generic parameters occur, and uses this information to invoke one version of the transformation or the other. 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 [52, 53] by presenting a more efficient representation, at the expense of carrying explicit type representations [25, 58]. *Minamide* further refines the transformation and is able to prove that the improved transformation does not affect the time complexity of the transformed program [34]. Tracking value representation in types has been presented and extended to continuation-passing style [20] by *Thiemann* in [59]. 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 if a certain representation is required. This information is used in a type inference algorithm which can elide conversions when the parameters are discarded or when a method call is in tail call position, namely it doesn't need to box the result only to have the caller unbox it. It should be noted that the conversions operate on a continuation-passing-style IR on the continuation-passing style.

A different direction in unboxing primitive types is based on escape analysis [17], where the program is analyzed at runtime and a conservative representation transformation is performed. When implemented in just-in-time compilers [54] of virtual machines such as PyPy [10] or Graal [64], and coupled with aggressive inlining, the escape analysis can make a big difference, possibly more than the global data representations presented in this paper. Still, these two techniques are fundamentally different – escape analysis has a local scope and relies heavily on inlining, while data representation transformation can safely optimize across method borders as long as the transformation will consistently make the same decisions in subsequent separate compilations. Interpreter-based techniques such as quickening [14] and trace-based specialization [21] can improve escape analysis with dynamic profiles of the program being executed. Truffle [63] 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 parametric polymorphism. Haskell exposes both the boxed `Int` representation and the unboxed `Int#`, but later transforms `Int` values to `Int#` by adding explicit conversions on each access. It then uses a peephole optimization [27, 32] to reduce all unnecessary boxing. The peephole optimizations have been formalized by *Henglein* in [26]. Haskell also features calling convention optimizations that make the argument laziness explicit and can unbox primitives in certain situations [9].

Value classes have been proposed for Java as early as 1999 [24, 47, 48]. The most recent description, which is also closest to our current approach, is the value class proposal for the Scala programming language [5]. 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 [36]. Value classes have also been implemented in the CLR [1], 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 [2] that, modulo the bottom type $\perp$, is unboxed similarly to value classes in Scala and CLR.

Specialization for generics is a technique aimed at eliminating boxing deep inside generic classes. Specialization has been implemented in Scala [18, 19] and has been improved by miniboxing [4, 60]. Specialization and macros have been combined to produce a mechanism for ad-hoc specialization of code in Scala [56]. The .NET framework automatically specializes all generics, thanks to its bytecode metadata and reified types [29].

A different approach to deep boxing elimination is described for Haskell [28] and Pyton [11]. 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) [57] requires lifting certain expressions in the program to a reified representation. Staging can be implemented using macros [15, 22] or using specialized compiler extensions [35]. 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) [13, 44, 46].

Annotated types [3, 6] 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 [40]. In the context of Java, type annotations have been used to selectively add reified type argument information to erased generics [23]. In the context of Scala, annotated types have been used to track and limit the side-effects of expressions [49, 50], to designate macro expansions [15] and to trigger continuation-passing-style transformations [45].

Formalization. In [33], *Leroy* presents a full formalization for the primitive unboxing for ML, including a proof of operational equivalence. The .NET generics are formalized in [65]. In the rules we state without a proof we rely on local type inference, as described by *Odersky et al.* [38] and *Pierce et al.* [42].

References

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

34yytbws.aspx.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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