Bridging Islands of Specialized Code using Macros and Reified Types

Nicolas Stucki
EPFL, Switzerland
nicolas.stucki@epfl.ch

Vlad Ureche
EPFL, Switzerland
vlad.ureche@epfl.ch

ABSTRACT

Parametric polymorphism in Scala suffers from the usual drawback of erasure on the Java Virtual Machine: primitive values are boxed, leading to indirect access, wasteful use of heap memory and lack of cache locality. For performance-critical parts of the code, the Scala compiler introduces specialization, a transformation that duplicates and adapts the bodies of classes and methods for primitive types. Specializing code can speed up execution by an order of magnitude, but only if the code is called from monomorphic sites or from other specialized code. Still, if these "islands" of specialized code are called from generic code, their performance becomes similar to that of generic code, losing optimality.
To address this, our project builds high performance "bridges" between "islands" of specialized code, removing the requirement that full traces need to be specialized: We use macros to delimit performance-critical "gaps" between specialized code, which we also specialize. We then use reified types to dispatch the correct specialized variant, thus recovering performance across the "islands". Our transformation[1] obtains speedups up to 30x and around 12x in average compared to generic only code, by enabling specialization to completely remove boxing and reach its full potential.

Categories and Subject Descriptors

D.2.3 [**Language Constructs and Features**]: Polymorphism; E.2 [**Object representation**]

General Terms

Data Representation, Specialization, Scala, Java virtual machine, Bytecode

Keywords

Specialization, Macros, Reified types

[1] `https://github.com/nicolasstucki/specialized`

Scala '13, Montpellier, France

1. INTRODUCTION

Whether ran in parallel or distributed across multiple nodes, the speed of sequential code directly influences the overall performance of the system. In particular, code using parametric polymorphism will be severely slowed down compared to monomorphic code when handling primitive values. The underlying problem is that primitive values come in different sizes and semantics, such as short or long integers, floating point numbers and characters. This conflicts with the uniform nature of parametric polymorphism, which assumes all objects have a common representation. The default translation in Scala aims at making primitive types uniform at the bytecode level by wrapping them into heap objects, in a process known as boxing. But boxing has several disadvantages, namely indirect access to values, which are located on the heap instead of the stack, wasteful use of memory by allocating redundant object headers along with values and lack of cache locality. This is a major concern for performance-critical code, and was addressed in the Scala compiler using the specialization transformation.

Specialization [3] is an annotation-driven transformation in the Scala compiler that improves performance of generic code. It is triggered by annotating a type parameter in a method or class, leading to the duplication and adaptation of the code body for each primitive type. Then, whenever a class is instantiated for a primitive type, the instantiation is rewritten to use the specialized variant instead of the generic one, avoiding boxing. The same rewriting is applied to method calls, where the invocation is redirected to the specialized variant.

In practice, specialization speeds up code execution by an order of magnitude [8]. But invoking the specialized code to benefit from this speedup remains difficult. Whenever a call site is either monomorphic (statically known to use a primitive type) or specialized, it is redirected to use the specialized variant of the method. Contrarily, a call site that uses a non-specialized type parameter is left pointing to the generic version of the method, since, thanks to erasure [1], there is no type information to dispatch the right specialized variant either at compile time or runtime. This creates little "islands" of specialized code, which can be invoked by monomorphic code to obtain performance, but revert to the non-specialized performance whenever they're called in a generic context.

Seen from a different angle, the "islands" of specialized code are traces in the program where the type information is encoded in the bytecode during the specialization transformation. But whenever the execution leaves the "island"

by calling generic code, the type information is lost, leading to the rest of the trace being generic and thus losing optimality. What's worse, parts of the generic trace may have specialized variants, but they're not used, since types are not available. Therefore, "islands" can call one another directly, but not indirectly via generic code. This breaks optimality in many important use cases.

The simplest solution for this problem is to specialize the generic code in between the "islands". Still, the amount of bytecode duplication makes this solution intractable. Furthermore, some of this code may reside in a library, thus making it impossible retroactively specialize. On the other hand, reified types [11] allow generic code to record types as values, making it possible to later use them to invoke specialized variants of the code.

This is where our project comes in: We bridge this gap between the "islands" by allowing the use of reified types to dispatch the correct specialized version. To this end, we allow the programmers to select a limited scope of generic code that will be specialized and which, in turn, will be able to call other specialized code. We then use reified types to dispatch on the correct specialized variant of the scope. This has two effects: we inject specialized "islands" in the middle of generic code, and these specialized "islands" can further call other specialized code, regaining optimality in the trace. Our project relies on macros and specialization to transform the scope and call the specialized variant.

In this context, we make the following two contributions: (i) we present a transformation that uses reified types to extend the scope of specialization; (ii) we validate our transformation by benchmarking it and obtaining speedups of up to 30x over Scala specialized code.

2. EXAMPLE

A situation where specialization is available but not used appears in the following example:

```
def createArray[T: Manifest](f: Int=>T) = {
  val a = new Array[T](size)
  for (i <- 0 until size)
    a(i) = f(i)
  a
}
```

Since the `createArray` method is not specialized, calling it either from generic or specialized code will yield the same slow operation: although both the array setter and the function call have specialized variants, the lack of specialization will mean the result of `f(i)` is boxed and the generic array setter is invoked. To make matters worse, the generic loop closures are not inlined by the Java Virtual Machine [6, 9], thus preventing escape analysis from eliminating boxing. But this is a perfect example where the `specialized` macro could kick in, since the reified type of `T` is available as a `Manifest`:

```
def createArray[T: Manifest](f: Int=>T) = {
  val a = new Array[T](size)
  specialized[T] {
    for (i <- 0 until size)
      a(i) = f(i)
  }
  a
}
```

In order to specialize the trace, the `specialized[T]` macro creates a new method which encodes the scope which needs to be specialized, which we will refer to it as the specialized body method. The specialized body method will be translated by the Scala compiler using the specialization transformation, which is triggered by the `@specialized` notation on type parameter `U`. This leads to the creation of multiple variants of `specBody`, one for each primitive type. Then, the last step is to invoke the correct variant.

```
def createArray[T: Manifest](f: Int=>T) = {
  val a = new Array[T](size)
  // specialized body method:
  def specBody[@specialized U]
       (a: Array[U], f: Int=>U) {
    for (i <- 0 until size)
      a(i) = f(i)
  }
  // dispatch code (without casts shown):
  manifest[T] match {
    case ManifestFactory.Int =>
      specBody[Int](a, f)
    ...
  }
  a
}
```

This leads to the rewriting of both the array setter and function `f` to use the specialized variants in each respective `specBody` variant, thus producing a completely specialized trace. The speedup obtained by the transformed code compared to the original one is 15x. On other examples, we obtain speedups of 5x to 30x.

We implemented the `specialized` macro [2] to wrap the code which needs to be specialized. This macro receives the polymorphic parameter, denoted by `T` in the example. It can also take a list of primitive types that will be specialized. If only `Int` and `Long` need fast execution, the programmer can use the following syntax:

```
specialized[T](Int, Long) { ... }
```

The dispatch consists of `match` statements, which carry some runtime overhead. Still, for the examples we have so far, dispatching a specialized version pays for the cost of dispatching. The next section will present the implementation.

3. IMPLEMENTATION

The `specialized` macro can be seen as a method that receives a closure and transforms the code inside it. It does so in the four steps shown in Figure 1. Each will be further explored in a separate subsection.

3.1 Checking Macro Call Parameters

The `specialized` macro takes four parameters: (i) the polymorphic type parameter `T` to be specialized; (ii) any number of `Specializable` objects, where `Specializable` is the trait that identifies the primitive types known to the compiler. In Scala these types are: `Int`, `Long`, `Boolean`, `Float`, `Double`, `Short`, `Char`, `Byte` and `Unit`; (iii) the main parameter is the block of code inside the curly braces. It consists of a closure of type `=>Any`; (iv) the last parameter is an implicit `Manifest` or `ClassTag`, which carries the reified type corresponding to `T` and is automatically filled in by the type checker.

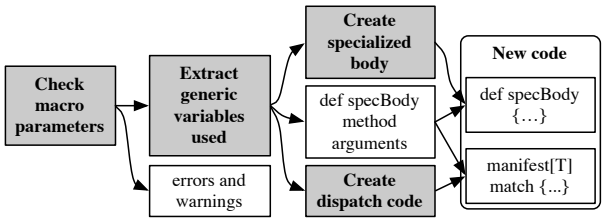

Figure 1: The code transformations taking place (in gray) and their results (in white).

Additionally there is a variant to handle the case where no `Specializable` parameters are set, which defaults to all primitive values. This implementation redirect to the first and main implementation. In an unambiguous context, where we have a single polymorphic type with manifest in scope, the macro can also be called without the first two parameters:

```
specialized { /* code goes here */ }
```

The code body to be specialized is given as a closure, which has a return type of `Any`. For a method this would be a problem, since it would mean we need to manually cast the result to its original type. But since our expansion takes place during the type checking phase, the replaced AST is type-checked again and a more precise return type is inferred. This allows us to sidestep the need to add a type parameter for the closure return, making both the macro implementation and its use simpler.

As soon as the macro is invoked, it checks if the input is correct. The type parameter `T` must be a bare polymorphic parameter: calls such as `specialized[Int]` or `specialized[List[T]]` will result in compile-time errors. It also checks that `T` has an accompanying manifest and that the block of code passed compiles and typechecks in its generic version. If any problem occurs it stops the transformation and shows the programmer an error corresponding to the problem. Other checks are also performed and can lead to warnings that guide the programmer to the correct use of the macro.

3.2 Extract Generic Variables Used

The second phase consists in extracting any references to generic variables used inside the block of code being specialized. In the `createArray` example on the previous page, the extracted variables are the array `a` and the function `f`. The array size, `size`, is not captured since its type is monomorphic (`Int`). The extracted variables will appear in the specialized body's signature.

3.3 Specialized Body Creation

The closure body appears in the `specBody` method. Here, the generic type `T` is replaced by a fresh polymorphic type parameter `U` which is marked as specialized. For the following example:

```
val (func: (T => T), seed: T) = ...
specialized[T](Int) {
  def rec(n: Int, last: T): T =
    if (n == 0) last
    else rec(n - 1, func(last))
  rec(1000000, seed)
}
```

The `specialized` macro transforms references from `T` to `U` deeply in the tree, including inside the `rec` method:

```
val (func: (T => T), seed: T) = ...
def specBody[@specialized(Int) U]
       (seed: U, func: U=>U) {
  def rec(n: Int, last: U): U = { ... }
  rec(1000000, seed)
}
(manifest[T] match {
  case ManifestFactory.Int =>
    specBody[Int](seed.asInstanceOf[Int],
                  func.asInstanceOf[Int=>Int])
  case _ => specBody[T](seed, func)
}).asInstanceOf[T]
```

The `U` type parameter receives the `@specialized` annotation so the Scala compiler is informed to duplicate and adapt it for the specific primitive values. The primitive `Specializable` values passed to the macro will further be passed on to the `@specialized(...)` annotation as arguments. In our example, `Int` is passed as the only `Specializable` argument. Along with duplicating the method, the specialization phase will also perform method call and class instantiation rewriting, making use of the specialized variantes where available. This creats new specialized traces.

As discussed previously, the specialized body will receive a list of generic parameters; these are determined in the previous step of the transformation and have their types rewired to reference the new polymorphic type `U`. The name of the specialized body method needs to be fresh to avoid any conflicts, but in the examples we use `specBody` for simplicity and conciseness.

The `specialized` macro works with typed Scala abstract syntax trees. After the namer and typer phases of the Scala compiler, when the macro receives the body to be specialized, trees have both types and symbols attached to each node. This means the body references the symbol of `T` and types containing `T`. To perform deep rewiring, once the `specBody` signature was created, the body undergoes a forced name resolution and typing to revisit all the tree nodes. This is done by first replacing references to the name `T` by the name `U` and then clearing the tree symbols and types. Since the resulting tree does not reference symbols and does not contain types, it undergoes another typing phase, which binds references to `U`.

3.4 Dispatching Code

The final step in rewriting the code is using the reified type to dispatch the correct specialized variant of the code body. To do so, we compare the type reification to each of the primitive types. If the comparison succeeds, we invoke the specialized body method with the correct type. The specialized method's parameters need explicit casting, since in the scope they are generic but in the invocation they have primitive types. Finally, we wrap the entire call to the specialized body method into a cast back to the original type, to make the types compatible with the generic code.

After the macro has transformed the code, another type checker run will fill in all the types and then the compiler will continue with the rest of its phases. Later, when the compiler reaches the specialization phase, the specialized body method will be duplicated and adapted. Once this is done, the compiler will rewrite the invocation in the code to one of the specialized variants.

4. EVALUATION

Using macros allowed us to add the transformation as a library method instead of a compiler plugin, which significantly lowers the adoption barrier. Thus our project makes no modification to the compiler and only requires an import statement to become available to programmers.

To evaluate the performance of this transformation we used ScalaMeter [10] to benchmark the execution time of the original and transformed code. The framework ran each time in a newly spawned HotSpot Java Virtual Machine and was warmed up before the actual benchmarking. We forced the JVM to use one of the following execution modes: interpreted only, compiled with client compiler (c1) only or compiled with server compiler (c2) only. This ensured our transformation speeds up execution in all execution states, making our transformed code consistently faster.

	`T`	Generic	Specialized	speedup
compiled (c1)	Int	24.72	1.83	13.51x
compiled (c2)	Int	24.57	1.97	12.47x
interpreted	Int	2458.29	255.86	9.61x
compiled (c1)	Double	19.12	2.06	9.28x
compiled (c2)	Double	19.22	1.96	9.81x
interpreted	Double	2668.26	269.71	8.89x
compiled (c1)	Boolean	22.49	3.40	6.61x
compiled (c2)	Boolean	22.67	3.13	7.24x
interpreted	Boolean	2040.98	266.75	7.65x
compiled (c1)	Generic	96.59	93.00	1.04x
compiled (c2)	Generic	93.89	92.87	1.01x
interpreted	Generic	2412.46	2336.18	1.03x

Table 1: Time in milliseconds necessary for executing the `createArray` example with an array of size 2 million.

We implemented algorithms that use generic instances of `Array`, `Function1`, `Function2` and `Tuple2` with some combinations between them or alone. We used code where there was at least one bridge between the specialized components. On operations over arrays the results yield speedups up to 14x specializing over `Int`, 20x specializing over `Double` and 30x specializing over `Boolean`. Table 1 and table 2 shows the speedups for the `createArray` example and a method that reverses an array.

	`T`	Generic	Specialized	speedup
compiled (c1)	Int	74.74	7.49	9.97x
compiled (c2)	Int	74.83	7.81	9.58x
interpreted	Int	5,241.05	387.47	13.53x
compiled (c1)	Double	68.71	8.91	7.71x
compiled (c2)	Double	69.18	8.84	7.82x
interpreted	Double	7,875.92	398.80	19.75x
compiled (c1)	Boolean	263.00	8.60	30.57x
compiled (c2)	Boolean	262.11	8.67	30.24x
interpreted	Boolean	6,084.31	406.01	14.99x
compiled (c1)	Generic	10.18	10.37	0.98x
compiled (c2)	Generic	10.26	10.46	0.98x
interpreted	Generic	2,886.12	3,109.43	0.93x

Table 2: Time in milliseconds necessary to reverse an array of size 10 million.

5. RELATED WORK

We build upon specialization [3], by adding the ability to use reified types. We also extend the range of code that can benefit from the specialization transformation: we add the ability to specialize scopes of code to the already existing abilities to specialize methods and classes.

Although the lack of global reified types significantly complicates the work of the specialization phase, their performance impact and memory footprint makes them undesirable in practice [11].

The .NET framework [5] is one example where specialization is greatly simplified by the existence of reified types. This happens for two reasons: reified types are implemented and optimized in the virtual machine, allowing for better handling. Also, the virtual machine provides hooks for runtime specialization, which allow just-in-time creation of specialized variants. This is hard to achieve in Java using the class loader mechanism [7], since it requires having full control over the running Java Virtual Machine.

Some JavaScript interpreters proposed trace specialization [4]. These require profiling and opportunistic trace transformations, that may need to be undone later if they prove too optimistic. In our case the static type system protects us from such cases, although we pay the cost of generating all the code up-front.

6. CONCLUSION

Our transformation is able to build high performance bridges between specialized code inside a generic context, allowing traces to be only partially specialized without losing performance. We use reified types dispatch the correct specialized implementaion.

The transformed code obtains speedups up to 30x and around 12x in average compared to generic only code, offering the full performance of specialization.

7. REFERENCES

[1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. *SIGPLAN Not.*, 33(10), Oct. 1998.

[2] E. Burmako. Scala Macros: Let Our Powers Combine! In *Proceedings of the 4th Annual Scala Workshop*, 2013.

[3] I. Dragos and M. Odersky. Compiling Generics Through User-Directed Type Specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, New York, NY, USA, 2009. ACM.

[4] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *ACM Sigplan Notices*, volume 44. ACM, 2009.

[5] A. Kennedy and D. Syme. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, New York, NY, USA, 2001. ACM.

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

[7] S. Liang and G. Bracha. Dynamic Class Loading in the Java Virtual Machine. *SIGPLAN Not.*, 33(10):36–44, Oct. 1998.

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

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

[10] A. Prokopec. ScalaMeter.

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