Semester project report

Bridging Islands of Specialized Code using Macros and Reified Types

Nicolas Stucki

EPFL

{first}.{last}@epfl.ch

Abstract

Parametric polymorphism in Scala suffers from the usual drawbacks of running 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. Specialized code can speed up execution by an order of magnitude, but only if they are called either from monomorphic sites or by other specialized code. But if these “islands” of specialized code are called from generic code, their performance becomes similar to generic code, losing its 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 “islands” of performance-critical code, and specialize it. 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 it’s full potential.

1. Introduction

Whether ran in parallel or distributed across multiple nodes, the speed of sequential code directly translates in 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 accessed 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 the 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 method invoked becomes 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 contributions:

- we present a transformation that makes use reified type to extend the scope of specialization
- we validate our transformation by benchmarking it and obtaining speedups of up to 30x compared to Scala specialization

The next section will introduce the running example that we’ll use throughout the paper to explain the transformation. The third

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

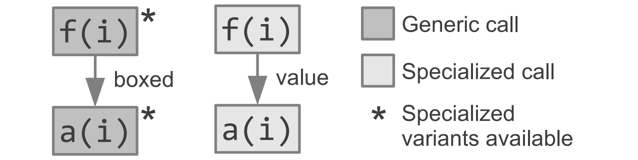

Figure 1. The calls made inside the loop before (*left*) and after (*right*) the specialized scope has been used.

section will briefly present the implementation. The forth section will present the benchmarks we ran on both original and transformed code. Finally, the fifth section will conclude.

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, as shown in left side of Figure 1. To make matters worse, the generic loop closures are not inlined by the Java Virtual Machine [6, 9], thus preventing other optimizations that could eliminate 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. 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) {
    // existing body:
    for (i <- 0 until size)
      a(i) = f(i)
  }

  // dispatch code:
  if (manifest[T] == ManifestFactory.Int) {
    // simplified call, without casts shown:
    specBody[Int](a, f)
  } else if ...

  a
}
```

This leads to both the array setter and `f` function to be rewritten to their specialized variants in each respective `specBody` variant, thus producing a completely specialized trace, shown in the right of Figure 1. The speedup obtained by the code at the bottom compared to the code at the top is 15x. Other examples we tried 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 reason the programmer might want to avoid specializing all primitive types is to eliminate some of the overhead of dispatching on the reified types. The dispatch consists of a long set of `if` 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 method is a macro that is able to rewrite the AST of the code that is inside it. It does so in the four steps shown in Figure 2. Each will be further explored in a separate subsection.

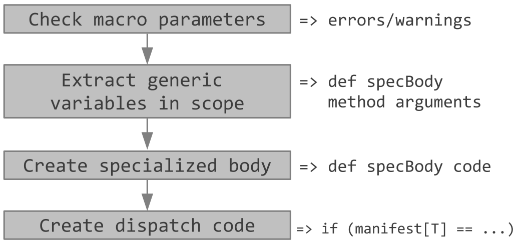

Figure 2. The code transformations taking place in the specialized macro and their results.

3.1 Checking Macro Call Parameters

The `specialized` macro takes four parameters:

- the polymorphic type parameter `T` to be specialized;
- 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`;
- the main parameter is the block of code inside the curly braces. It consists of a closure of type `=>Any`;
- 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.

Therefore the macro is defined as:

```
def specialized[T](types: Specializable*)
      (expr_f: => Any)
      (implicit classTag: ClassTag[T]): Any
  = macro impl_specialized[T]
```

Additionally there are two other variants to handle the case where no `Specialized` parameters are set (this defaults to all primitive values) and one to handle `SpecializedGroup`. Those implementations redirect to the first and main implementation. In an unambiguous context, where we have a single polymorphic type with a manifest in scope, the macro can also be called without the first two parameters:

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

As can be seen from the signature, the closure's expected return type is `Any`, 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 another type parameter for the return type, making both the macro implementation and its use easier.

As soon as the macro is invoked, it checks if the input is correct. It checks that the parameter `T` is in fact a polymorphic parameter, issuing errors in obviously incorrect calls such as `specialized[Int]` or `specialized[List[T]]`. It also checks that `T` has a manifest and that the block of code inside it compiles and typechecks under 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 Variables Used in the Scope

The second phase consists in extracting generic variables used in the scope. We first identify any references to variables inside the block of code being specialized. In the `createArray` example, the variables are the array `a` and the function `f`. A more selective extraction happens in the example below, where only `array` is extracted. This happens because `array` is of type `Array[T]`, therefore is generic. But `size` is not extracted, since its type is monomorphic (`Int`):

```
  val size = array.length
  specialized[T] {
    // reversing the array, using size
  }
```

This gives the signature of the method that encloses the specialized body.

3.3 Specialized Body Creation

The specialized body method, which corresponds to the `specBody` method in the expanded `createArray` example, will contain the code that was passed to the `specialize` macro. Instead of using the generic type `T`, it will specialize over a new polymorphic type parameter `U`, such that the specialization phase can kick in. Therefore the macro rewrites all references from type `T` to `U`.

The `U` type parameter receives the `@specialized` annotation so the Scala compiler is informed to duplicate and adapt it for the primitive values. The primitive values passed to the macro will further be passed on to the `@specialized` annotation. Along with duplicating the method, the specialization phase will also perform method call and class instantiation rewriting, making use of the specialized code available and creating new specialized traces. This is where the transition in Figure 1 happens.

To illustrate the transformation, we will use the following code:

```
val func: T => T = ...
val seed: T = ...
specialized[T] {
  def rec(n: Int, last: T): T =
    if (n == 0) last
    else rec(n - 1, func(last))
  rec(2000000, seed)
}
```

The work needed for the creation of the specialized body method consists in rewiring any explicit `T` to `U` before putting it in the specialized body method, including the parameters. This requires clearing all the symbols and types of the abstract syntax tree, such that a type checking phase binds them to the new definitions that refer to the `U` type parameter. An example where all this is needed is the one where we compose n times a function and apply it to a first element (`seed` in this case). Note that `rec` is a tail recursive function, meaning that without the macro it would box and unbox the parameter each iteration of the loop.

The specialized body method definition will also need the list of parameters it receives; those are the ones that were identified in step two. The types of these arguments will also be rewired to the new polymorphic type `U` explicitly. The last consideration needed is that the name of the specialized body method needs to be fresh to avoid any conflicts, in the examples we wrote `specBody` for simplicity and conciseness. This is the final `specBody` code:

```
def specBody[@specialized U]
        (seed: U, func: U=>U) {
  def rec(n: Int, last: U): U = { ... }
  rec(2000000, seed)
}
```

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 reified type to each of the primitive types. If the comparison succeeds, we invoke the specialized body method with the correct type:

```
if (manifest[T]==ManifestFactory.Int) {
  // invoke the specialized body for Int
  ...
} else if (manifest[T]==ManifestFactory.Long)
  // invoke the specialized body for Long
  ...
} else {
  // invoke the generic body
  specBody[T](seed, func)
}
```

Once we know that we can simply force the specialized body method to use the specialized version by casting all it's parameters (the ones identified in the second step). 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. In the following example, this is shown for `Int`:

```
  // invoke the specialized body for Int
  specBody[Int](seed.asInstanceOf[Int],
       func.asInstanceOf[Int=>Int])
```

After the macro has transformed the code, the type checker will refill in all the types and then the compiler will keep on going normally. Only 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. This is the final missing link in our use of reified types for dispatching specialized variants of code.

4. Evaluation

By generating the transformation using macros we where able add it as a simple library method. This means no modification to the compiler was needed and that a single import is necessary. This makes the tool extremely simple to use.

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

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

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 different VM and was warmed up before the actual benchmarking. We forced each time the compiler to use only one of the following modes: interpreted only, compiled with client compiler (c1) only or compile with server compiler (c2) only. This enabled us to ensure our transformation speeds up execution in all execution modes of the HotSpot JVM, making our transformed code consistently faster.

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

Table 2. Time in milliseconds necessary for executing the array reversal example with an array of size 10 million.

We implemented algorithms that use generic `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` and `reverseArray` examples. Table 3 shows the speedups for the last code example in which we compose a function several times over numeric values.

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.

	T	Original	Specialized	speedup
compiled with c1	Int	50.24	2.26	22.24x
compiled with c2	Int	50.32	2.26	22.24x
interpreted	Int	4,349.99	655.01	6.64x
compiled with c1	Double	35.00	14.37	2.44x
compiled with c2	Double	34.59	14.50	2.39x
interpreted	Double	7,270.90	654.54	11.11x

Table 3. Time in milliseconds necessary for executing the function composition example, applying the function `(x => 42*x)` 10 million times.

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 they require having control of the entire Java stack.

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 project was able to build high performance bridges between specialized code inside a generic context, allowing traces to be only partially specialized without changing their characteristics. We use reified types to identify the different types at runtime allowing us to force the execution of specialized code.

Overall, our transformation obtained speedups up to 30x and around 12x in average compared to generic only code, by offering the full performance of specialization and reducing unnecessary boxing.

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. ISSN 0362-1340.

[2] E. Burmako. Scala macros. URL `http://scalamacros.org/`.

[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. ISBN 978-1-60558-541-3.

[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. ISBN 1-58113-414-2.

[6] 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.

[7] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. *SIGPLAN Not.*, 33(10):36–44, Oct. 1998. ISSN 0362-1340. doi: 10.1145/286942.286945. URL `http://doi.acm.org/10.1145/286942.286945`.

[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. URL `http://axel22.github.com/scalameter/`.

[11] M. Schinz. *Compiling scala for the Java virtual machine*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

[12] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages*. ACM, 2012.