École Polytechnique Fédérale de Lausanne

Semester Project - LAMP

Report

Implementing Method Type Specialisation In Dotty

Author:
Alexandre Sikiaridis

Supervised by:
Dmitry Petrashko
Martin Odersky

05/06/2015

Contents

Chapter 1

Introduction

Type Specialisation is an optimisation feature allowing efficient use of generics. It is therefore an important feature of statically typed languages, and adding support for it in Dotty was a step that had to be taken eventually.

This report first briefly recalls what type specialisation is. It compares the compilation result (using scala 2.11.6) of a function making use of parametric polymorphism with and without type specialisation, and discusses results.

The implementation of method type specialisation in Dotty is then described. It is broken down into four parts, each of which is detailed out. Major issues encountered are then explained, as well as the consequences they had on design choices.

Finally, a road map of future developments is outlined.

Chapter 2

Parametric Polymorphism & Type Specialisation

Parametric polymorphism is an important feature in statically typed languages, as it allows efficient writing of generic code, without forsaking static typing richness. For instance, defining a `head` method on lists of any type can be summarised to a single function, instead of defined for each possible type:

```
def head[A](list: List[A]): A = list(0)

instead of

def head_Int(list: List[Int]): Int = list(0)
def head_Char(list: List[Char]): Char = list(0)
...
```

2.1 Boxing and Unboxing

By default, parametric polymorphism is compiled away by the scala compiler through erasure. This means type parameters are erased and replaced by their upper-bound - `Object` by default. For primitive types to adapt to that, they have in turn to be *boxed* as `Objects`.[1]
The following code snippet proposes a generic method, and its compilation result (using scala 2.11.6):

```
object Test {
  def checkWithDefault[T](elem: T, p: T => Boolean, default: T): T = if (p(elem)) elem else
     default
  checkWithDefault[Int](5, _<4, 0)
```

```
4 }
```

```
1  package <empty> {
2    object Test extends Object {
3      def checkWithDefault(elem: Object, p: Function1, default: Object): Object = if
       (scala.Boolean.unbox(p.apply(elem)))
4        elem
5      else
6        default;
7      def <init>(): Test.type = {
8        Test.super.<init>();
9        Test.this.checkWithDefault(scala.Int.box(5), {
10         (new <$anon: Function1>(): Function1)
11       }, scala.Int.box(0));
12       ()
13     }
14   };
15   @SerialVersionUID(value = 0) final <synthetic> class anonfun$1 extends
       scala.runtime.AbstractFunction1$mcZI$sp with Serializable {
16     final def apply(x$1: Int): Boolean = anonfun$1.this.apply$mcZI$sp(x$1);
17     <specialized> def apply$mcZI$sp(x$1: Int): Boolean = x$1.<(4);
18     final <bridge> <artifact> def apply(v1: Object): Object =
       scala.Boolean.box(anonfun$1.this.apply(scala.Int.unbox(v1)));
19     def <init>(): <$anon: Function1> = {
20       anonfun$1.super.<init>();
21       ()
22     }
23   }
24 }
```

As expected, the integer values `5` and `0` are *boxed* at lines 9 and 11 in order to be passed to the function as an `Object`. Once the function receives `elem`, it passes it to the `p` predicate function (line 3), which needs to unbox the value, compute the boolean result, box that again and return it (line 18). `checkWithDefault` then has to unbox that result again to a Boolean (line 3), and use it to determine its return value.
A lot of boxing and unboxing - which can have a high impact on performance.
It is interesting to notice that specialisation is already involved in this example: the anonymous function `_ < 4` has been translated by the compiler to specialised variants of `AbstractFunction1` and `apply`, as illustrated by the `$mcZI$sp` suffixes.

2.2 Specialisation

Specialisation is a technique aimed at solving parametric polymorphism's efficiency issues. In order to avoid for methods to go through the process of boxing and unboxing

values, the compiler creates variants of each method where generic parameters are replaced by primitive types. Here is the same code as before, with specialisation activated on `Int`s:

```
1  object Test {
2    def checkWithDefault[@specialized(Int) T](elem: T, p: T => Boolean, default: T): T = if
       (p(elem)) elem else default
3    checkWithDefault[Int](5, _<4, 0)
4  }
```

```
1  package <empty> {
2    object Test extends Object {
3      def checkWithDefault(elem: Object, p: Function1, default: Object): Object = if
       (scala.Boolean.unbox(p.apply(elem)))
4        elem
5      else
6        default;
7      <specialized> def checkWithDefault$mIc$sp(elem: Int, p: Function1, default: Int): Int = if
       (p.apply$mcZI$sp(elem))
8        elem
9      else
10       default;
11
12   // Same as before
13   // ...
14
15   }
16 }
```

The effect of specialisation is here obvious, as lines 7 to 10 now define a new function `checkWithDefault$mcI$sp`, which no longer takes `Object`s as parameters, but `Int`s, and as hoped, no longer needs to box and unbox values.
The drawback of specialisation is apparent as well: the generated code's size has grown. Had the specialized annotation not been completed with "`(Int)`", specialised variants of `checkWithDefault` would have been generated for all primitive types (`Int`, `Short`, `Long`, `Double`, `Float`, `Char`, `String`, `Boolean`, `Unit`), resulting in nine extra function definitions. And had it been a function with two generic types, this could have amounted to 81 more functions generated. This exponential growth is specialisation's main drawback, as it creates a tradeoff between significant code size blow-up if not careful, and efficient running times.

2.3 A Word on Miniboxing

A word concerning Miniboxing is of interest here.[2]
Miniboxing is an alternative to specialisation, aimed at getting the best of both worlds by ensuring the run-time efficiency of specialisation whilst avoiding any code blow-up.

It is based on the simple realisation, that at the low-level perspective of the JVM, there are only values and pointers - primitive types are not distinguished. Using this fact, a rough explanation is that miniboxing encodes types into long integers, and thus only produces 2^n new methods instead of 9^n for normal specialisation.
That being said, miniboxing also adds a lot of complexity; something which was decided unnecessary as specialisation in Dotty works only on-demand.

Chapter 3

Method Type Specialisation in Dotty

3.1 Scheme

Method type specialisation in Dotty works on-demand. It follows the current scala syntax.

3.2 Breakdown of Functionality

Specialisation of methods in Dotty can be broken down into four components:

- Retrieval of `@specialized` annotations
- Generation of new symbols
- Generation of trees based on the new symbols
- Dispatching of the new methods to appropriate call sites

3.2.1 Annotations Retrieval

Specialisation in Scalac is triggered by the use of the `@specialized` annotation, and the same scheme is used in Dotty. Therefore, the first task of specialisation is the retrieval of those annotations; for reasons discussed below (3.3.1) this is done in a separate phase called `PreSpecializer`.

Its implementation relies on the following code:

```
def getSpec(sym: Smbol)(implicit ctx: Context): List[Type] = {
    if (allowedToSpecialize(sym)) {
      val annotation = sym.denot.getAnnotation(defn.SpecializedAnnot).getOrElse(Nil)
      annotation match {
        case annot: Annotation =>
          val args = annot.arguments
          if (args.isEmpty) primitiveTypes
          else args.head match {
            // Matches simple `@specialized(...)` annotations
            case a @ Typed(SeqLiteral(types), _) =>
              types.map(t => primitiveCompanionToPrimitive(t.tpe))

            // Matches `@specialized` annotations on Specializable Groups
            case a @ Ident(groupName) if a.tpe.isInstanceOf[Type] =>
              specializableToPrimitive(a.tpe.asInstanceOf[Type], groupName)

            case _ => ctx.error("unexpected match on specialized annotation"); Nil
          }
        case nil => Nil
      }
    } else Nil
  }
```

The call to `allowedToSpecialize(sym)` ensures that no specialisation happens on methods that should not be so - which includes `isInstanceOf` and `asInstanceOf` methods, Java defined methods, and constructors (because of design decisions in Dotty, specialisation of constructors creates errors further down the pipeline.)
Looking for specialisation annotations then eventually leads to five possible cases:

1 - an `@specialized` annotation with no parameters, in which case specialisation is activated on all primitive types.

2 - an `@specialized` annotation with primitive types as parameters, in which case specialisation is activated on all the concerned primitive types.

3 - an `@specialized` annotation with `AnyRef` given as parameter, in which case specialisation is activated on `AnyRef`. This case is distinguished, as an `AnyRef` could not be handled exactly the same way as a primitive type.

4 - an `@specialized` annotation where parameters are groups of the `Specializable` trait. In this case, types of the group are gathered from a mapping instantiated during an ovverriding call to `prepareForUnit`.

5 - no `@specialized` annotation, in which case specialisation is not activated.

All the specialisation types that are thus gathered are passed to the `TypeSpecializer` phase through a `PhaseCache` called `specializePhase`:

```
1   ctx.specializePhase.asInstanceOf[TypeSpecializer]
2     .registerSpecializationRequest(tree.symbol)(types)
```

3.2.2 Symbols Generation

Once the methods to specialise have been determined, it is up to the ‘TypeSpecializer‘ phase to take care of generating and using the specialised methods appropriately.
The first step of the process is the generation of symbols representing those methods. This is done during an overriding call to `transformInfo`.
Types found in annotations are fetched, combinations of specialised types are created by the `generateSpecializations` method, and corresponding specialised symbols are generated by the `generateSpecializedSymbols` method. Those symbols are stored in a map from generic method to specialized variants and their list of types.

```
1  def generateSpecializations(remainingTParams: List[Name], specTypes: List[Type])
2                 (instantiations: List[Type], names: List[String], poly: PolyType, decl: Symbol)
3                                 (implicit ctx: Context): List[Symbol] = {
4      if (remainingTParams.nonEmpty) {
5        specTypes.map(tpe => {
6          generateSpecializations(remainingTParams.tail, specTypes)
7             (tpe :: instantiations, specialisedTypeToSuffix(ctx)(tpe) :: names, poly, decl)
8        }).flatten
9      }
10     else {
11       generateSpecializedSymbols(instantiations.reverse, names.reverse, poly, decl)
12     }
13   }
14 def generateSpecializedSymbols(instantiations: List[Type], names: List[String],
                                                              poly: PolyType, decl: Symbol)
15                                 (implicit ctx: Context): List[Symbol] = {
16     val newSym =
17       ctx.newSymbol(decl.owner, (decl.name + "$mc" + names.mkString + "$sp").toTermName,
18         decl.flags | Flags.Synthetic, poly.instantiate(instantiations.toList))
19     val map = newSymbolMap.getOrElse(decl, mutable.HashMap.empty)
20     map.put(instantiations, newSym)
21     newSymbolMap.put(decl, map)
22     map.values.toList
23   }
```

3.2.3 Trees Rewriting

With the symbols generated, ‘TypeSpecializer‘ next overrides ‘transformDefDef‘ to generate variants of all ‘DefDef‘s. They are then passed on as a `Thicket`.

The specialised symbols generated previously are first fetched, and a new `PolyDefDef` is created for each.

The `treeTypeMap` instance which is passed as parameter to that `PolyDefDef` has a `treeMap` argument which is built to make calls to `transformApply`, which will take care of inner method dispatching. The `typeMap` argument will substitute generics and type parameters of arguments as defined by the specialised symbol.

Casts are also introduced here, through the calls to `ensureConforms(...)`. Their necessity is discussed in the issue regarding type information below.

```
1  polyDefDef(newSym.asTerm, { tparams => vparams => {
2    val tmap: (Tree => Tree) = _ match {
3      case Return(t, from) if from.symbol == tree.symbol => Return(t, ref(newSym))
4      case t: TypeApply => transformTypeApply(t)
5      case t: Apply => transformApply(t)
6      case t => t
7    }
8
9    val typesReplaced = new TreeTypeMap(
10     treeMap = tmap,
11     typeMap = _
12       .substDealias(origTParams, instantiations(index))
13       .subst(origVParams, vparams.flatten.map(_.tpe)),
14     oldOwners = tree.symbol :: Nil,
15     newOwners = newSym :: Nil
16     ).transform(tree.rhs)
17
18     val tp = new TreeMap() {
19     // needed to workaround https://github.com/lampepfl/dotty/issues/592
20     override def transform(t: Tree)(implicit ctx: Context) = super.transform(t) match {
21       case t @ Apply(fun, args) =>
22         assert(sameLength(args, fun.tpe.widen.firstParamTypes))
23         val newArgs = (args zip fun.tpe.widen.firstParamTypes).map{case(t, tpe) =>
24                                                                    t.ensureConforms(tpe)}
25         if (sameTypes(args, newArgs)) {
26           t
27         } else tpd.Apply(fun, newArgs)
28       case t: ValDef =>
29         cpy.ValDef(t)(rhs = if (t.rhs.isEmpty) EmptyTree else t.rhs.ensureConforms(t.tpt.tpe))
30       case t: DefDef =>
31         cpy.DefDef(t)(rhs = if (t.rhs.isEmpty) EmptyTree else t.rhs.ensureConforms(t.tpt.tpe))
32       case t => t
33     }}
34     val expectedTypeFixed = tp.transform(typesReplaced)
35     expectedTypeFixed.ensureConforms(newSym.info.widen.finalResultType)
36 }
```

3.2.4 Method Dispatching

The specialised variants are finally dispatched to call sites, according to inferred types. Overriding calls to `transformApply` and `transformTypeApply` replace method calls with specialised variants whenever possible. Parameterless `TypeApply` instances are specialized as such, whereas `Apply` instances need extra casts to be added to their arguments.
The choice of the specialised variant to apply is based on checking that all expected argument types of the method are subtypes of the variant in question.

3.3 Issues Encountered

3.3.1 Lost Annotations

One of the first major issues encountered regarded annotations.
As explained, specialisation relies on Types annotated with `@specialized`, and is triggered only then. `TypeSpecializer`'s first task was therefore initially to check all `DefDef`s for annotations, and specialise them accordingly.
However, because annotations are not stored in Types but in Trees, they cannot be retrieved by `TypeSpecializer` itself.
Suppose the following code:

```
1 val s = foo[Int]
2 def foo[@specialized T]: T
3 val d = foo[Double]
```

When `TypeSpecializer` finishes transforming `s`, it has not yet read the definition of `foo`, and so cannot know in advance that it should be specialised and `s` be dispatched to the appropriate variant.
To avoid the issue, the selected solution was to have another phase - `PreSpecializer` - run earlier in the pipeline, gather the necessary information, and store it in a *PhaseCache* for the later-running `TypeSpecializer` phase.

3.3.2 Loss of Type Information

As described earlier, method specialisation in Dotty relies on casts when the typechecker cannot infer suitable types.

Consider the following:

```
trait A {
  type T1
  type T2
  def foo[B <: T1 >: T2] = ...
}
```

In this instance, it would seem safe to assume that `T2 <:< T1` in the body of the method. The following snippet of code exhibits a similar case:

```
trait Foo[@specialized +A] {
  def bop[@specialized B >: A]: Foo[B] = this
}
```

which should compile to:

```
def bop_sp[@specialized B >: A <: Int & Any]: Foo[B] = this
```

`this` has type `Foo[A]`, and based on the knowledge that `B >: A`, it seems clear that `Foo[A] <:< Foo[B]`. However, the typechecker does not infer such a result currently, leading to errors.

The introduction of casts appears therefore unavoidable in all places where expected types could lead to such issues. The following snippet illustrates a few:

```
trait Foo[@specialized +A] {
// all those examples trigger bugs due to https://github.com/lampepfl/dotty/issues/592
  def bop[@specialized B >: A]: Foo[B] = new Bar[B](this)
  def gwa[@specialized B >: A]: Foo[B] = this
  def gwd[@specialized B >: A]: Foo[B] = {
    val d: Foo[B] = this
    d
  }
}
case class Bar[@specialized a](tl: Foo[a]) extends Foo[a]
```

This issue has been reported to the github repository as issue #592. (https://github.com/lampepfl/dotty/issues/592)

3.3.3 Multiple Specialisation Options

As described, when dispatching specialised methods, the choice of which specialised variant to use is based on the inferred type of the function, and arguments to the function being subtypes of the variant's argument types. While this seems reasonable, there exists a corner case: what if the expected type is `Nothing`? Then all specialised variants will match, and there is *à priori* no way of selecting one over the other.
In such cases, Dotty will default to not specialising.

```
1 object nothing_specialization {
2   def ret_nothing[@specialized(Char) T] = {
3     def apply[@specialized(Char) X](xs : X*) : List[X] = List(xs:_*)
4     def apply6[@specialized(Char) X](xs : Nothing*) : List[Nothing] = List(xs: _*)
5 }
```

3.3.4 Super Calls

Calls to `super()` will not be specialised with the current implementation. The reason for that is `super` methods are generated as early as `PosTyper`, which does not pass annotations onto them. `PreSpecializer` has therefore currently no way of finding them.

Issue #631 has been opened in the repository in this matter. (https://github.com/lampepfl/dotty/issues/631)

3.4 Testing

Several tests were written to check for those various features, and issues. While avoiding errors is done in pos tests, checking that code is generated and used as it should be is tested in run tests.

3.4.1 Pos Tests

Several tests have been written to check for compilation errors. Those check for correct specialisation in the following cases:

- simple methods
- mutually recursive methods

- inner methods (both with regard to an outer method or an outer class)
- methods with multiple generic types
- methods requiring the introduction of casts in order to typecheck
- methods using different @specialized annotations (no argument, type arguments, Specializable Group argument)

3.4.2 Run Tests

A run test has been designed, checking for the generation of the correct amount of specialised variants, and with the correct type parameters.

Chapter 4

Future Developments

We go over the next implementation steps necessary before type specialisation is complete in Dotty.

4.1 Partial Method Specialisation

Currently, method specialisation is implemented fully - that is, for all type parameters of the method. In the following, both `U` and `T` would therefore be specialised :

```
object Test {
  def foo[@specialized T, U](t: T): T = ???
}
```

such that Dotty would output the following tree after specialisation:

```
package <empty> {
  final lazy module val Test: Test$ = new Test$()
  final module class Test$() extends Object() { this: Test.type =>
    def foo[@specialized() T, U](t: T): T = ???
    def foo$mcCC$sp(t: Char): Char = ???
    def foo$mcZC$sp(t: Boolean): Boolean = ???
    ... (78 other methods)
    def foo$mcVB$sp(t: Unit): Unit = ???
  }
}
```

Perhaps however is specialising only `T` sufficient to the writer's goal. Partial method specialisation would allow that, while keeping `U` as a generic type; Dotty would then output something along the lines of the following:

```
package <empty> {
  final lazy module val Test: Test$ = new Test$()
  final module class Test$() extends Object() { this: Test.type =>
    def foo[@specialized() T, U](t: T): T = ???
    def foo$mcC$sp[U](t: Char): Char = ???
    def foo$mcZ$sp[U](t: Boolean): Boolean = ???
    ... (6 other methods)
    def foo$mcV$sp[U](t: Unit): Unit = ???
  }
}
```

As of the writing of these lines, implementation of partial specialisation is partially realised, but still contains too many bugs to be included in a pull request.

4.2 Class Specialisation

Once partial method specialisation is complete, specialisation will have to be extended to classes, as described in [1].

Bibliography

[1] I. Dragos. Type specialization in scala 2.8, May 2010. URL http://www.scala-lang.org/old/sites/default/files/sids/dragos/Thu,%202010-05-06,%2017:56/sid-spec.pdf.

[2] V. Ureche C. Talau, M. Odersky. Miniboxing: Improving the speed to code size tradeoff in parametric polymorphism translations. *OOPSLA '13*, 2013.