Code walkthrough of the SpecializeTypes phase (Scala 2.8)

© Miguel Garcia, STS, Hamburg University of Technology
http://www.sts.tu-harburg.de/people/mi.garcia

November 17, 2009

DRAFT

Contents

1 Where to start **1**

2 SpecializeTypes **2**

3 Input-output example **4**
- 3.1 Without type specialization 5
- 3.2 With type specialization 5

4 Debugging: first encounter with the code structure **5**

Abstract

A previous walkthrough (for the CleanUp phase) explored some aspects of the inner workings of the Scala compiler, showing IDE tips all along. Little was said about how type information is held in AST nodes. In order to gain *some* (read: partial) insight into that topic, this walkthrough focuses on some aspects of the `specialize` phase, originally described in *Compiling Generics Through User-Directed Type Specialization* [1] and in the Scala SID 9 [1]. The following notes serve as an introduction to the implementation of the phase, biased to newcomers. Accomplished Scala compiler hackers won't find new insights in these notes: you might consider contributing a phase description of your own, or extend this description, thanks!

1 Where to start

One may or may not know that `specialize` is deactivated by default, requiring `-Yspecialize` as command-line option to `scalac`. And given that type specialization is all about speed, `-optimize` also makes sense. BTW, that setting is a.k.a. `XO` in `Settings.scala`.

Armed with that knowledge, one can search in `Global.scala` for code realizing that compiler phase, i.e.,

[1] http://www.scala-lang.org/sid/9

```
// phaseName = "specialize"
object specializeTypes extends {
  val global: Global.this.type = Global.this
  val runsAfter = List[String]("")
  val runsRightAfter = Some("tailcalls")
} with SpecializeTypes
```

Pressing `ALT + F7` on `specializeTypes` shows where this phase is conditionally added in `computerInternalPhases()`:

```
  if (settings.specialize.value)
    phasesSet += specializeTypes
```

2 SpecializeTypes

The implementation of the `specialize` phase stretches over `SpecializeTypes.scala` and `TypingTransformers.scala`, with `InfoTransform` being part of the `nsc` infrastructure for compilation phases (Figure 1). Let's take a look at the first file, containing the definition of `SpecializeTypes`:

`abstract class SpecializeTypes extends InfoTransform with TypingTransformers`

The recipe when implementing a compiler phase appears to be:

1. If implementing an (external, custom) compiler plugin phase, then subclass `Plugin`, which makes available handy functionality (such as `processOptions`, whose comment reads: "Handle any plugin-specific options. The `-P:plugname:` part will not be present"). Built-in phases skip this step in the recipe.
2. nested in the above, subclass `SubComponent` (or one of its subclasses).
3. depending on whether `Transform` was subclassed in the previous item, either nest in that subclass a subclass of `Transformer` or `Phase`.

   - An example of the above recipe is `http://github.com/jsuereth/private-setter-scalac-plugin` described at `http://suereth.blogspot.com/2009/02/levaraging-annotations-in-scala-part-2.html`
   - For example, the `CleanUp` phase extends `Transform` and contains a subclass of `Transformer`, instantiated by `newTransformer`
   - Otherwise (as in `Pickler`) subclass `Phase` and override `newPhase` to return an instance of it.
   - Want to cross-check? Take a look at Figure 2
   - Coming back to `SpecializeTypes`, it extends `InfoTransform` and defines the following transformer:

```
class SpecializationTransformer(unit: CompilationUnit) extends Transformer {
  override def transform(tree: Tree) =
    atPhase(phase.next) {
      val res = specializeCalls(unit).transform(tree)
      res
    }
}
```

DRAFT

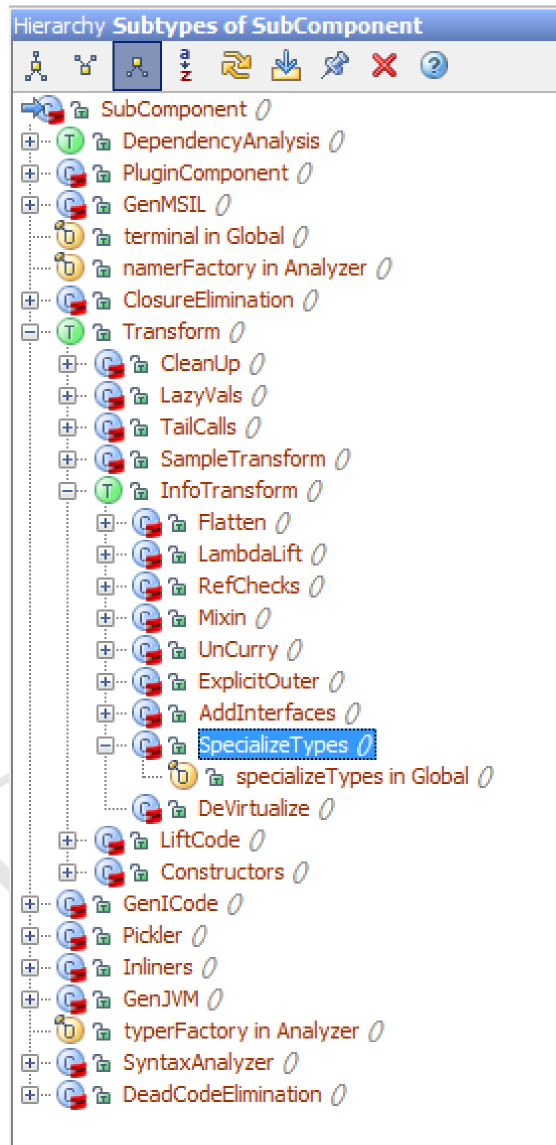

Figure 1: Different kinds of compilation phases (i.e., subclasses of `SubComponent`, not of `Phase`)

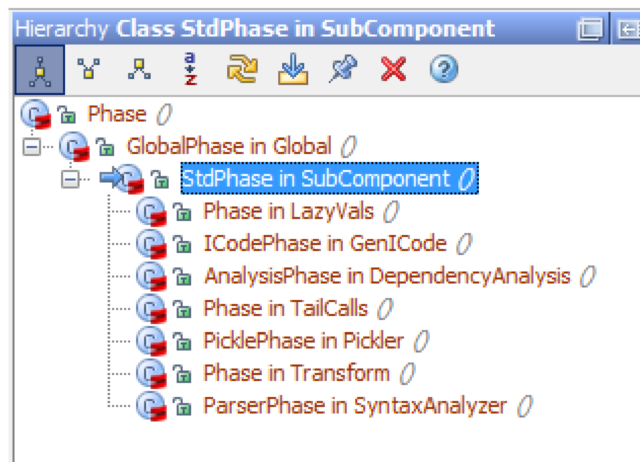

Figure 2: Different kinds of phases (related but not equivalent to compilation phases)

> As well as a `TypingTransformer` (in the second file in the implementation, `TypingTransformers.scala`). Each of these two transformers overrides `transform(tree: Tree): Tree` (the method that does the real work). Which transformer will prevail? Wait and see.

Besides the very popular `Transformer` and `Traverser`, other utilities defined nearby in `Gloabl.scala` include: `StrictTreeCopier, LazyTreeCopier, TreeSubstituter, TreeTypeSubstituter, TreeSymSubstituter, ChangeOwnerTraverser`. Perhaps someone should write about them too.

Talking about utilities, it's possible to provide a custom `AnnotationChecker` to achieve the following:

```
/** An additional checker for annotations on types.
 *  Typically these are registered by compiler plugins
 *  with the addAnnotationChecker method. */
```

If you know examples of checkers like that in action, please let me know :-)

3 Input-output example

Before delving into the phase implementation we'll see what to expect from it, in terms of a simple input-output example, reproduced from a discussion about the performance of generic code when manipulating arrays[2]

From all the examples in that post, we'll take the simplest: reversing an array (Listing 1). The whole point of the example is the performance penalty incurred without specialization. Quoting from one of the replies in the thread:

> The problem is that you're using polymorphic arrays. In Scala, it means that it has to work with primitive arrays (`int[]`, `long[]`, etc.)

[2] `http://old.nabble.com/Arrays%3A-Disappointed-about-ridiculous-slow-scala-code-compared-to--Java-to26193962.html`

Listing 1: Our testbed (!)

```
def reverse[@specialized("Int") T](a: Array[T]) {
  var lo = 0
  var hi = a.length - 1
  while (lo < hi) {
    val tmp = a(lo)
    a(lo) = a(hi)
    a(hi) = tmp
    lo += 1
    hi -=1
  }
}
```

> as well as boxed arrays (`Object[]`, `String[]`, etc.). The end result is a nasty performance penalty. In Java, polymorphic arrays are sort of broken (try `new T[0]`) and they don't try to unify primitives with boxed representations so the performance penalty is not there.

BTW, Scala 2.8 arrays are the topic `http://www.scala-lang.org/sid/7`.

3.1 Without type specialization

After running `nsc` with `-Xprint:jvm` (Listing 2), the definition of `array_apply` in `runtime.this.ScalaRunTime` reveals that indeed boxing/unboxing will occurr:

```
/** Retrieve generic array element */
def array_apply(xs: AnyRef, idx: Int): Any = java.lang.reflect.Array.get(xs, idx)
```

3.2 With type specialization

After activating the `specialize` phase, the output looks as shown in Listing 3.

The example does not reveal any of the fine points about type specialization (for example those mentioned in the Future Work section of [1]). That's our running example, though :-)

4 Debugging: first encounter with the code structure

Before debugging proper it helps to see what the input right before the `specialize` phase is, in our case with `-Xprint:tailcalls`. Starting with the breakpoint shown in Figure 3, one is taken first to the instantiation of `TypingTransformer` with its in-place refinement (whose structure is not shown in the Structure View, and there's some structure to it as it stretches between lines 855 and 1215):

```
def specializeCalls(unit: CompilationUnit) = new TypingTransformer(unit) {
    . . . here comes the in-place refinement
```

which involves initializing `Transformer` (and by consequence `currentOwner`, and by consequence `currentMethod`, `currentClass`, and `currentPackage`), and `localTyper`,

DRAFT

Listing 2: What the compiler produces without type specialization

```
[[syntax trees at end of jvm]]// Scala source: reverse.scala
package <empty> {
  final class rev extends java.lang.Object with ScalaObject {
    def reverse(a: java.lang.Object): Unit = {
      var lo: Int = 0;
      var hi: Int = runtime.this.ScalaRunTime.array_length(a).-(1);
      while$1(){
        if (lo.<(hi))
        { {
          val tmp: java.lang.Object = runtime.this.ScalaRunTime.array_apply(a, lo);
          runtime.this.ScalaRunTime.array_update(a, lo, runtime.this.ScalaRunTime.array_apply(a, hi));
          runtime.this.ScalaRunTime.array_update(a, hi, tmp);
          lo = lo.+(1);
          hi = hi.-(1)
          };
          while$1()
        }
        else
          ()
      }
    };
    def this(): object rev = {
      rev.super.this();
      ()
    }
  }
}
```

DRAFT

and three maps that perhaps are important so I'll note them down next (although they are not contained in the same lexical scope, they are shown one after the other):

```
    /** a typer for each enclosing class */
    var typers: Map[Symbol, analyzer.Typer] = new HashMap

    /** Map a specializable method to it's rhs, when not deferred. */
    val body = new mutable.Map[Symbol, Tree]

    /** Map a specializable method to its value parameter symbols. */
    val parameters = new mutable.Map[Symbol, List[List[Symbol]]]
```

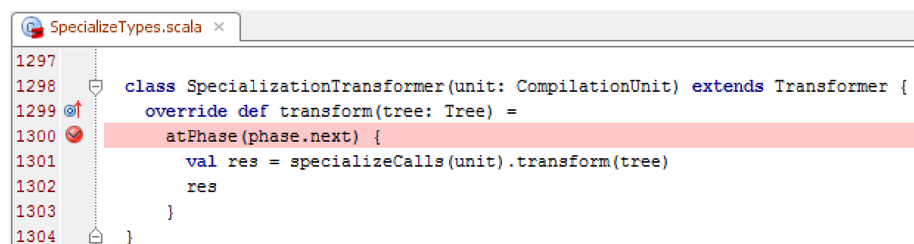

Figure 3: First breakpoint in our tour

Listing 3: What the compiler produces without type specialization

```
[[syntax trees at end of jvm]]// Scala source: reverse.scala
package <empty> {
  final class rev extends java.lang.Object with ScalaObject {
    def reverse(a: java.lang.Object): Unit = {
      ... as above
    };
    def reverse$mIc$sp(a: Array[Int]): Unit = {
      var lo: Int = 0;
      var hi: Int = a.length().-(1);
      while$1(){
        if (lo.<(hi))
          {
            {
              val tmp: Int = a.apply(lo);
              a.update(lo, a.apply(hi));
              a.update(hi, tmp);
              lo = lo.+(1);
              hi = hi.-(1)
            };
            while$1()
          }
        else
          ()
      }
    };
    def this(): object rev = {
      rev.super.this();
      ()
    }
  }
}
```

DRAFT

Now that we have three layers (`Transformer`, `TypingTransformer`, and its refinement) you may think it's a good time to gain some perspective on their workings, peeling off one at a time so to say. Problem is, in order to get a feeling for `Transformer`, one needs to have a feeling for tree matching ("A standard pattern match" in `Trees.scala`). And so on. But yes, after that kind of deep diving one finally comes back to where we started.

After getting out of the activation for `specializeCalls(unit)` we get into that of `transform(tree)`, whose definition ranges from lines 882 to 1060, but the main case match inside it goes only from line 904 to line 1059. For a bird's eye view see Listing 4.

Strolling through the case patterns in Listing 4 makes for moments which demand a hefty load of previous knowledge, as for example to understand this comment:

```
    case PackageDef(pid, stats) =>
      tree.symbol.info // make sure specializations have been peformed
      ...
```

Listing 4: What the compiler produces without type specialization

```
/*  905 */ case Apply(Select(New(tpt), nme.CONSTRUCTOR), args) =>


/*  915 */ case TypeApply(Select(qual, name), targs)
           if (!specializedTypeVars(symbol.info).isEmpty && name != nme.CONSTRUCTOR) =>


/*  934 */ case Select(qual, name) if (/*!symbol.isMethod
                                 &&*/ !specializedTypeVars(symbol.info).isEmpty
                                 && name != nme.CONSTRUCTOR) =>

/*  957 */ case PackageDef(pid, stats) =>

/*  965 */ case Template(parents, self, body) =>

/*  971 */ case ddef @ DefDef(mods, name, tparams, vparamss, tpt, rhs)
           if info.isDefinedAt(symbol) =>

/* 1046 */ case ValDef(mods, name, tpt, rhs) if symbol.hasFlag(SPECIALIZED) =>


/* 1057 */ case _ =>
```

References

[1] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47, New York, NY, USA, 2009. ACM.

DRAFT