Changes in Compiler Plugins

Edit this page on GitHub

Compiler plugins are supported by Dotty (and Scala 3) since 0.9. There are two notable changes compared to `scalac`:

- No support for analyzer plugins
- Added support for research plugins

Analyzer plugins in `scalac` run during type checking and may influence normal type checking. This is a very powerful feature but for production usages, a predictable and consistent type checker is more important.

For experimentation and research, Scala 3 introduces *research plugin*. Research plugins are more powerful than `scalac` analyzer plugins as they let plugin authors customize the whole compiler pipeline. One can easily replace the standard typer by a custom one or create a parser for a domain-specific language. However, research plugins are only enabled for nightly or snaphot releases of Scala 3.

Common plugins that add new phases to the compiler pipeline are called *standard plugins* in Scala 3. In terms of features, they are similar to `scalac` plugins, despite minor changes in the API.

Using Compiler Plugins

Both standard and research plugins can be used with `scalac` by adding the `-Xplugin:` option:

```
scalac -Xplugin:pluginA.jar -Xplugin:pluginB.jar Test.scala
```

The compiler will examine the jar provided, and look for a property file named `plugin.properties` in the root directory of the jar. The property file specifies the fully qualified plugin class name. The format of a property file is as follows:

```
pluginClass=dividezero.DivideZero
```

This is different from `scalac` plugins that required a `scalac-plugin.xml` file.

Starting from 1.1.5, `sbt` also supports Scala 3 compiler plugins. Please refer to the `sbt` documentation for more information.

Writing a Standard Compiler Plugin

Here is the source code for a simple compiler plugin that reports integer divisions by zero as errors.

```
package dividezero

import dotty.tools.dotc.ast.Trees.*
import dotty.tools.dotc.ast.tpd
import dotty.tools.dotc.core.Constants.Constant
import dotty.tools.dotc.core.Contexts.Context
import dotty.tools.dotc.core.Decorators.*
import dotty.tools.dotc.core.StdNames.*
import dotty.tools.dotc.core.Symbols.*
import dotty.tools.dotc.plugins.{PluginPhase, StandardPlugin}
import dotty.tools.dotc.transform.{Pickler, Staging}

class DivideZero extends StandardPlugin:
  val name: String = "divideZero"
  override val description: String = "divide zero check"

  def init(options: List[String]): List[PluginPhase] =
    (new DivideZeroPhase) :: Nil

class DivideZeroPhase extends PluginPhase:
  import tpd.*

  val phaseName = "divideZero"

  override val runsAfter = Set(Pickler.name)
  override val runsBefore = Set(Staging.name)

  override def transformApply(tree: Apply)(implicit ctx: Context): Tree =
    tree match
      case Apply(Select(rcvr, nme.DIV), List(Literal(Constant(0))))
      if rcvr.tpe <:< defn.IntType =>
        report.error("dividing by zero", tree.pos)
      case _ =>
        ()
```

```scala
    tree
end DivideZeroPhase
```

The plugin main class (`DivideZero`) must extend the trait `StandardPlugin` and implement the method `init` that takes the plugin's options as argument and returns a list of `PluginPhase` s to be inserted into the compilation pipeline.

Our plugin adds one compiler phase to the pipeline. A compiler phase must extend the `PluginPhase` trait. In order to specify when the phase is executed, we also need to specify a `runsBefore` and `runsAfter` constraints that are list of phase names.

We can now transform trees by overriding methods like `transformXXX` .

Writing a Research Compiler Plugin

Here is a template for research plugins.

```scala
import dotty.tools.dotc.core.Contexts.Context
import dotty.tools.dotc.core.Phases.Phase
import dotty.tools.dotc.plugins.ResearchPlugin

class DummyResearchPlugin extends ResearchPlugin:
  val name: String = "dummy"
  override val description: String = "dummy research plugin"

  def init(options: List[String], phases: List[List[Phase]])(implicit ctx: Cont
    phases
end DummyResearchPlugin
```

A research plugin must extend the trait `ResearchPlugin` and implement the method `init` that takes the plugin's options as argument as well as the compiler pipeline in the form of a list of compiler phases. The method can replace, remove or add any phases to the pipeline and return the updated pipeline.

‹ Autom... Lazy V... ›