INTEGRATING IDEs WITH DOTTY

Guillaume Martres - EPFL
WHAT IS DOTTY?

- Research compiler that will become Scala 3
- Type system internals redesigned, inspired by DOT, but externally very very similar
- More info:
  - dotty.epfl.ch
  - Recent blog posts on scala-lang.org
A CHANCE TO REDESIGN COMPONENTS

- Improved incremental compilation (avoid undercompilation)
- Better pattern matching checks (algorithm now reused in Swift!)
TOOLING

A good developer experience requires good tools:

- A REPL (with syntax highlighting!)
- Dottydoc (used to generate [dotty.epfl.ch/docs](dotty.epfl.ch/docs))
- IDE support
STATE OF THE ART

• Based on the Scala Presentation Compiler
  ▪ Scala-IDE
  ▪ ENSIME
• Reimplementation of the Scala typechecker
  ▪ Scala plugin for IntelliJ IDEA
STATE OF THE ART

- Based on the Scala Presentation Compiler (3 KLOC)
  - Scala-IDE (66 KLOC)
  - ENSIME (server: 15 KLOC, emacs client: 10 KLOC)
- Reimplementation of the Scala typechecker
  - Scala plugin for IntelliJ IDEA (230 KLOC)
DESIGN PRINCIPLES

1. Code reuse
2. Editor-agnosticity
3. Easy to use (and to install!)
DESIGN PRINCIPLES

1. Code reuse
2. Editor-agnosticity
3. Easy to use (and to install!)
• Each phase progressively simplify trees until they can be emitted as JVM bytecode
Each phase progressively simplify trees until they can be emitted as JVM bytecode
final val elem = 1

val foo = elem + 1

- Cursor position = 🟢
- Query: jump to definition
Every tree node has a type and a position

Query can be answered

```scala
final val elem: Int = 1
val foo: Int = elem + 1
```
TREE AFTER FirstTransform

final val elem: 1 = 1

val foo: Int = 2

- Information lost by constant folding
- Impossible to answer query
QUERYING THE COMPILER

- Store trees before `FirstTransform`
- Respond to IDE queries by traversing trees
- What about code that has already been compiled?
• In Scala 2: store methods signatures (for separate compilation)
• In Dotty: store full trees
TASTY: TYPED AST SERIALIZATION FORMAT

• Original motivation: solve the binary compatibility problem
  ▪ Always use JVM bytecode: breaks when compiler encoding changes
  ▪ Always recompile source code: breaks when the typechecker changes
• Can also be used to provide interactive features: deserialize and query trees
INTERACTIVE APIS

• Convenience methods for tree traversals, compiler lifecycle management
• Used both in the IDE and the REPL (e.g., for completions)
• In the future: interruption handling, partial typechecking, ...
• Less then 1 KLOC
DESIGN PRINCIPLES

1. Code reuse
2. Editor-agnosticity
3. Easy to use (and to install!)
THE IDE PORTABILITY PROBLEM

Getting $m$ IDEs to support $n$ programming languages requires $n \times m$ IDE plugins.
THE LANGUAGE SERVER PROTOCOL

Language Server Protocol

Tool

Language Server

User opens document

User edits document

Language server sends errors/warnings

User executes “Go To Definition”

User closes document

textDocument/didOpen: textDocument

textDocument/didChange: textDocument

textDocument/publishDiagnostics: diagnostics

textDocument/definition: textDocument, position

result: uri, range

textDocument/didClose: textDocument

JSON RPC
BASICS OF THE LSP

- First implemented in Visual Studio Code
- JSON-RPC
- IDE notifies the language server about user actions
- LS maintains internal representation of code
- LS notify IDE about warnings/errors
- IDE can send requests usually triggered by user actions
- Asynchronous, cancelable
IMPLEMENTING THE DOTTY LANGUAGE SERVER

- Low-level message handling done by Eclipse LSP4J
- Relies on interactive APIs
- 0.5 KLOC
override def definition(params: TextDocumentPositionParams) =

}
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>

}
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getDocumentUri)}
}
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
  }
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx
  }

override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx

    val pos = sourcePosition(driver, uri, params.getPosition)
  }
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx

    val pos = sourcePosition(driver, uri, params.getPosition)
    val uriTrees = driver.openedTrees(uri)
  }

}
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx

    val pos = sourcePosition(driver, uri, params.getPosition)
    val uriTrees = driver.openedTrees(uri)
    val sym = Interactive.enclosingSourceSymbol(uriTrees, pos)
  }
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx

    val pos = sourcePosition(driver, uri, params.getPosition)
    val uriTrees = driver.openedTrees(uri)
    val sym = Interactive.enclosingSourceSymbol(uriTrees, pos)

    val classTree =
      SourceTree.fromSymbol(sym.topLevelClass.asClass).toList
  }
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx

    val pos = sourcePosition(driver, uri, params.getPosition)
    val uriTrees = driver.openedTrees(uri)
    val sym = Interactive.enclosingSourceSymbol(uriTrees, pos)

    val classTree =
      SourceTree.fromSymbol(sym.topLevelClass.asClass).toList
    val defTree = Interactive.definition(classTree, sym)
  }
override def definition(params: TextDocumentPositionParams) =
  computeAsync { cancelToken =>
    val uri = new URI(params.getTextDocument.getUri)
    val driver = driverFor(uri)
    implicit val ctx = driver.currentCtx

    val pos = sourcePosition(driver, uri, params.getPosition)
    val uriTrees = driver.openedTrees(uri)
    val sym = Interactive.enclosingSourceSymbol(uriTrees, pos)

    val classTree =
      SourceTree.fromSymbol(sym.topLevelClass.asClass).toList
    val defTree = Interactive.definition(classTree, sym)
    defTree.map(d => location(d.namePos)).asJava
  }
DESIGN PRINCIPLES

1. Code reuse
2. Editor-agnosticity
3. Easy to use (and to install!)
SBT INTEGRATION

- Analyze the build to find Dotty projects
- Compile these projects
- Generate configuration files
- Install the Dotty VSCode extension
- Launch VSCode
CONFIGURATION FILES

- `.dotty-ide-artifact`, used by the IDE extension to launch the Dotty Language Server:

  `ch.epfl.lamp:dotty-language-server_0.8:0.8.0-RC1`
CONFIGURATION FILES

- `.dotty-ide.json`, used by the DLS to launch compiler instances:

```json
[
  {
    "id": "root/compile",
    "compilerVersion": "0.8.0-RC1",
    "compilerArguments": [],
    "sourceDirectories": [ "src/main/scala" ],
    "dependencyClasspath": [ ... ],
    "classDirectory": "target/scala-0.8/classes"
  },
  {
    "id": "root/test",
    ...
  },
  ...
]
```
BUILD SERVER PROTOCOL

• Instead of making plugins for build tools to extract information, ask them!
• We also need a discovery protocol: "How do I start a build server for this project?"
1. Code reuse
   • Compiler APIs for interactive usage
2. Editor-agnosticity
   • Implemented the LSP
3. Easy to use (and to install!)
   • One command (but we can do better!)
GOING FURTHER: DEBUGGER SUPPORT

- Based on the Java Debug Server
- Most features "just work"
- (Not actually merged in Dotty yet)
- Challenge: expression evaluation
class Hello {
    def foo(implicit y: Context): String = { /*...*/ }
    def bar(implicit y: Context) = {
        /*...*/
    }
}
class Hello {
    def foo(implicit y: Context): String = {
    /*...*/
    }
    def bar(implicit y: Context) = {
    /*...*/
    }
}
class Hello {
    def foo(implicit y: Context): String = { /*...*/ }
    def bar(implicit y: Context) = {
        foo
        /*...*/
    }
}
class Hello {
    def foo(y: Context): String = {
        /*...*/
    }
    def bar(y: Context) = {
        this.foo(y)
        /*...*/
    }
}
object Global {
    def liftedExpr($this: Hello, y: Context) =
        $this.foo(y)
}
object Global {
    def liftedExpr($this: Hello, y: Context) =
        $this.foo(y)

    def exec(self: Object, localVariables: Map[String, Object]) =
        liftedExpr(
            self.asInstanceOf[Hello],
            localVariables("y").asInstanceOf[Context]
        )
}
ON THE DEBUGGING VM

- Compile Global to a classfile
- Load it in the debugged VM
- Call Global.exec with the right arguments
ON THE DEBUGGED VM

- In the standard library

```scala
package dotty.runtime

object DebugEval {
  def eval(classpath: String,
           self: Object,
           localVariables: Map[String, Object]): Any = {
    val cl = new URLClassLoader(Array(new URL("file://" + classpath))
    val cls = cl.loadClass("Global")
    val instance = cls.newInstance
    val exec = cls.getMethod("exec")
    exec.invoke(instance, self, names, args)
  }
}
```
ON THE DEBUGGING VM

We want to remotely execute:

```scala
val classpath = <classpath for the compiled Global class>
val self = <this in the stackframe>
val localVariables = <map of local variables in the stackframe>
dotty.runtime.DebugEval(classpath, self, localVariables)
```
ON THE DEBUGGING VM

We use the Java Debugging Interface APIs:

```scala
val vars = stackFrame.visibleVariables
val mapCls = 
    vm.classesByName("java.util.Map")
    .get(0).asInstanceOf[ClassType]
val localVariablesRef = mapCls.newInstance()
// Skipped: store `vars` into `localVariablesRef`
val debugCls = 
    vm.classesByName("dotty.runtime.DebugEval")
    .get(0).asInstanceOf[ClassType]
val eval = 
    debugCls.methodsByName("eval").get(0)
debugCls.invokeMethod(
    thread, eval,
    List(vm.mirrorOf(classpath), stackFrame.thisObject,
    localVariablesRef)
)
```
FUTURE WORK

- Optimizations
- More features
  - Documentation on hover
- Better build tool integration (Build Server Protocol!)
CONCLUSION

- Design your compiler with interactivity in mind
- Design your build tool with interactivity in mind
- Interactivity should go beyond what IDEs and REPLs currently offer
  - Type Driven Development with Idris
QUESTIONS?

- More info: dotty.epfl.ch
- Come chat with us: 
gitter.im/lampepfl/dotty
- Contributors welcome!