1 A closer integration in the Scala compiler

In the file `tangle.nw`, we outlined on how one could puzzle together a compileable file out of a stream of blocks. This is very useful on its own, but a closer compiler integration is desirable: This way, we can tell the compiler the real line numbers of our literate program and not of the tangled source - we will be able to debug more efficiently. Also, we can access information provided by the compiler: What was defined by this block and what variables will be used.

Concretely, we will be implementing `CoTangle` which will call the compiler with the code taken from the stream of blocks. `LitComp` is the command line application that directly compiles an input literate program.

Additionally, we provide an object that directly exposes `LiterateProgramSourceFiles`. This way, we can even use `scalac` to work with literate programs.

⟨*⟩ ≡

```
package scalit.tangle
<CoTangle – send tangled to compiler>
<A source file format for literate programs>
<A new position type>
<LitComp – the command line application>
<LiterateCompilerSupport – object for scalac>
```

1.1 CoTangle

The following class will pass the source files that we get to the compiler (and maybe a destination directory), so a first step is to include the compiler class. Later, we'll outline how to actually compile.

⟨*CoTangle - send tangled to compiler*⟩ ≡

```
class CoTangle(sourceFiles: List[LiterateProgramSourceFile],
               destination: Option[String]) {
  <Include a compiler>
  <Compile a literate program>
}
```

The compiler will also have to have a place to report errors to, so we will need a reporter. The usual way would be to have an object overriding some behavior for this, but we will just instantiate a standard reporter.

⟨*Include a compiler*⟩ ≡

```
import scala.tools.nsc.{Global,Settings,reporters}
import reporters.ConsoleReporter
```

At the moment, there is only one setting that we might give to the compiler: If we have a destination directory, then we'll pass it

⟨*Include a compiler*⟩ + ≡

```
val settings = new Settings()
destination match {
  case Some(dd) ⇒ {
    settings.outdir.tryToSet(List("−d",dd))
  }
  case None ⇒ ()
}
val reporter =
  new ConsoleReporter(settings,null,
                                        new java.io.PrintWriter(System.err))
val compiler = new Global(settings,reporter)
```

1.2 Literate program as a source file

After we have instanciated a compiler, we want to feed him a source file. In this file, we'd like to conserve original line numbering etc.

Much of the functionality will be close to a batch source file. The main thing that changes is the mapping position → position in literate file.

⟨*A source file format for literate programs*⟩ ≡

```
import scala.tools.nsc.util.{BatchSourceFile,Position,LinePosition}
class LiterateProgramSourceFile(chunks: ChunkCollection)
  extends BatchSourceFile(chunks.filename,
                                        chunks.serialize("*").toArray) {
  <line mappings>
  <find a line>
  <the original source file>
  <position in ultimate source file>
}
```

1.3 Position in ultimate source file

The tangling process rearranges lines from the original literate program so that they are a valid source input. For compiling purposes, we want to find out to which original line a line in the tangled file corresponds. As this will be done quite often, we want to cache the results:

⟨*line mappings*⟩ ≡

```
  val lines2orig = new scala.collection.mutable.HashMap[Int,Int]()
```

Finding a line amounts to iterating over the stream of code blocks until we find the one containing the line. This might not be an optimal solution - now that we are eating the first newline after the chunk definition, it might actually happen that we point to the wrong line, for example with code like

```
val s = <Read s>
```

and

⟨⟨`Read s`⟩⟩=

```
1/0
```

Here, the error is in the second part of the code, but the line corresponding to the first part will be returned. One way to fix this would be to consider offset information.

⟨*find a line*⟩ ≡

```
import scalit.markup.StringRefs._
lazy val codeblocks: Stream[RealString] =
  chunks.expandedStream("*")

def findOrigLine(ol: Int): Int =
  if( lines2orig contains ol ) lines2orig(ol)
  else {
    def find0(offset: Int,
          search: Stream[RealString]): Int = search match {
      case Stream.empty ⇒ error("Could not find line for " + ol)
      case Stream.cons(first,rest) ⇒
        first match {
          case RealString(cont,from,to) ⇒ {
            val diff = to − from
            if( ol ≥ offset && ol ≤ offset + diff ) {
              val res = from + (ol − offset)
              lines2orig += (ol → res)
              res
            } else
              find0(offset + diff,rest)
          }
        }
    }

    find0(0,codeblocks)
  }
```

Most of the work was already done in the tangling phase, so we can just check whether we are inside a string from the source file. For error reporting, we will want to point to the original source file, but there is, of course a problem: The source might come from a markup file, or standard input and not just a literate program. But in any case except reading a literate program from standard input (which is of dubious utility anyway), we know the original source file because of the `@file` directive, which is then given to us as an argument. This is very suboptimal: We slurp the whole file for random access:

⟨*the original source file*⟩ ≡

```
import scala.tools.nsc.util.{SourceFile,CharArrayReader}
lazy val origSourceFile = {
  val f = new java.io.File(chunks.filename)
  val inf = new java.io.BufferedReader(
    new java.io.FileReader(f))
  val arr = new Array[Char](f.length().asInstanceOf[Int])
  inf.read(arr,0,f.length().asInstanceOf[Int])
  new BatchSourceFile(chunks.filename,arr)
}
```

With all this information, we can finally override the method that tells us the position in the original source file. To access the original source file,

⟨*position in ultimate source file*⟩ ≡

```
override def positionInUltimateSource(position : Position) = {
  val line = position.line match {
    case None ⇒ 0
    case Some(l) ⇒ l
  }
  val col = position.column match {
    case None ⇒ 0
    case Some(c) ⇒ c
  }
  val literateLine = findOrigLine(line)
  LineColPosition(origSourceFile,literateLine,col)
}
```

The position class that we return is also defined especially for this use - we do not want to count the offset into the literate file:

⟨*A new position type*⟩ ≡

```
import scala.tools.nsc.util.SourceFile
case class LineColPosition(source0: SourceFile, line0: Int,
                           column0: Int) extends Position {
  override def offset = None
  override def column: Option[Int] = Some(column0)
  override def line: Option[Int]   = Some(line0)
  override def source = Some(source0)
}
```

1.4 The compilation process

With the source file format in place, calling the compiler becomes quite simple: We create a new `Run` which will compile the files:

⟨*Compile a literate program*⟩ ≡

```
  def compile: Global#Run = {
    val r = new compiler.Run

    r.compileSources(sourceFiles)
    if( compiler.globalPhase.name != "terminal" ) {
      System.err.println("Compilation failed")
      System.exit(2)
    }

    r
  }
```

At the moment, we are not very specific about error reporting: If compilation does not work, we'll just exit.

1.5 The command line application

For this command line application, we just take the list of chunks from the literate settings. Then fo every such chunk we'll create a source file. All of these source files are compiled together and stored in the path given by the `-d` command line flag.

⟨*LitComp - the command line application*⟩ ≡

```
object LitComp {
  def main(args: Array[String]): Unit = {
    import scalit.util.LiterateSettings

    val ls = new LiterateSettings(args)

    val sourceFiles = ls.chunkCollections map {
      cc ⇒ new LiterateProgramSourceFile(cc)
    }

    val destinationDir: Option[String] =
      ls.settings get "−d" match {
        case Some(x :: xs) ⇒ Some(x)
        case _ ⇒ None
      }

    val cotangle = new CoTangle(sourceFiles,
                                destinationDir)

    cotangle.compile
  }
}
```

1.6 Support for scalac

When the `scalac` compiler is invoked, it works directly on `SourceFiles`. Before these are parsed, we have to map the literate file to one of concrete code. Fortunately, we already have a class `LiterateProgramSourceFile` which serves that purpose. We therefore only provide one function, taking the filename of the literate source, building the chunks and then returning a `LiterateProgramSourceFile`.

⟨*LiterateCompilerSupport - object for scalac*⟩ ≡

```
object LiterateCompilerSupport {
  def getLiterateSourceFile(filename: String): BatchSourceFile = {
    import scalit.util.conversions
    val cbs = conversions.codeblocks(conversions.blocksFromLiterateFile(filename))
    val chunks = emptyChunkCollection(filename) addBlocks cbs
    new LiterateProgramSourceFile(chunks)
  }
}
```