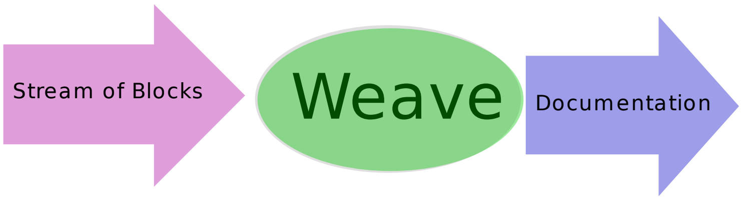
1 Weave - Creating documentation out of a literate program

The program described here allows us to extract a human readable file out of the `noweb` input. The content will be output in the order that it was written in the noweb file, but code sections will be annotated and we will gather information that allows us to indicate information like which class was defined where and so on.



It is important to note that there exists a tool to do syntactic highlighting for scala called `verbfilter`[1]. The method `process` takes a character buffer and looks for `\begin{verbatim}`, there it will begin to transform the input. Our aim is therefore to convert the code sections to a character buffer so that it can be fed to verbfilter.

⟨*⟩ ≡

```
package scalit.weave
import markup._

<The LaTeX weaver>

<Source file information>

<The command line application>
```

1.1 The LaTeX weaver

LatexWeaver, the class that takes care of producing LaTeX output from a list of streams of blocks (as defined in `weave/blocks.nw`) has two methods: One for printing one block, `printBlock`, and one to print everything surrounding to produce a valid LaTeX document.

⟨*The LaTeX weaver*⟩ ≡

[1] To be found under misc/scala-tool-support/latex

```
sealed abstract class Weaver(blocks: List[(Stream[Block],String)])

case class LatexWeaver(blocks: List[(Stream[Block],String)],
                       tangled: List[tangle.ChunkCollection],
                       useVerbfilter: Boolean,
                       useIndex: Boolean,
                       classpath: Option[List[String]],
                       filename: String,
                       useHeader: Boolean)
    extends Weaver(blocks) {

  import java.io.PrintStream

  <escape in quoted sections>

  <escape code>

  <print one block>

  <print the document>

  <compiler help>

  <format index>
}
```

The useVerbfilter flag tells the weaver whether it should fire up the compiler to retreive information on the source code. This is made optional because it is pretty expensive. If `useHeader` is set to false, then we will not print a header (ideal for inclusion in other LaTeX files).

1.2 Printing one block

There are two types of blocks to consider, code and documentation blocks. In documentation blocks, we will need to know how to escape content: The backslash character especially has to be escaped.

⟨*escape in quoted sections*⟩ ≡

```
def escape(orig: String): String = {
  orig.replace("\\","$\\backslash$")
      .replace("{","$\\{$")
      .replace("}","$\\}$")
      .replace("#","$\\#$")
      .replace("<","$\\langle$")
      .replace(">","$\\rangle$")
      .replace("_","\\_")
}
```

This function will undoubtedly be extended by more escape sequences. Now on how to actually print these blocks. One speciality that we want to indicate is whether we have already begun the definition of a given chunk. As this would be too cumbersome to carry around, we'll keep track of it here:

$\langle$*print one block*$\rangle \equiv$

```
val chunksSeen =
  new scala.collection.mutable.HashMap[String,CodeBlock]
```

On to the block printing:

$\langle$*print one block*$\rangle + \equiv$

```
import StringRefs._
def printBlock(out: PrintStream,
       chunks: tangle.ChunkCollection)
       (b: Block): Unit = b match {
  case cb @ CodeBlock(bn,ln,content,blockname) => {
    out.print("$\\left<\\mbox{\\emph{" +
        blockname +
        "}}\\right>")

    if( chunksSeen contains blockname ) out.print("+")
    else chunksSeen += (blockname -> cb )

    out.println("\\equiv$")
```

Chunks that get defined for the first time start with $\langle$*name*$\rangle \equiv$, a continued chunk will be of the form $\langle$*name*$\rangle + \equiv$. If we use other code chunks, this will be noted by ⟨*name*⟩. In the following section we will slurp the whole content of the code block into a string:

⟨*print one block*⟩ + ≡

```
var begin = -1
var end   = -1
val content = "\\begin{verbatim}" +
((cb.stringRefForm(chunks.cm) map {
    case RealString(cont,from,to) => {
        if( begin == -1 ) begin = from

        if( to > end ) end = to

        cont
    }
    case BlockRef(b) => {
        "<" +
        b.blockname +
        ">"
    }
    case other => error("Unexpected: " + other)
} foldLeft "") {
    (acc: String, next: String) => acc + next
}) + "\\" + "end{verbatim}"
```

If we want to use verbfilter, then we'll pass it to the script. As we will have some unescaping to do (especially the $ character is troublesome) the output is not directly sent to out but stored in a byte array, on which we can then apply the unescaping. Also, if we want to create an index, we'll have to tell it here.

⟨*print one block*⟩ + ≡

```
    if( useVerbfilter ) {
      val vfOutput = new java.io.ByteArrayOutputStream
      toolsupport.verbfilterScala
        .process(codeEscape(content).getBytes,vfOutput)
      if( useIndex )
        out.println(
          indexed(
          codeUnescape(
          vfOutput.toString),
          begin,end,chunks.filename))
      else
        out.println(codeUnescape(vfOutput.toString))
    } else {
      if( useIndex )
        out.println(indexed(content,
          begin,end,chunks.filename))
      else
        out.println(content)
    }
  }
```

Here we just avoided a quine-like problem: `\end{verbatim}` is the sequence to terminate a code block, so if it occurs inside a code block, then we could run into a problem.

Documentation blocks will contain escaped sections (quoted), but otherwise they will be copied verbatim.

⟨*print one block*⟩ + ≡

```
case d @ DocuBlock(bn,ln,content) ⇒ {
  d.stringRefForm(Map()) foreach {
    x ⇒ x match {
      case RealString(cont,_,_) ⇒ out.print(cont)
      case QuotedString(cont) ⇒ {
        out.print("\\texttt{")
        out.print(escape(cont))
        out.print("}")
      }
      case BlockRef(_) ⇒
        error("Did not expect code reference" +
        " in documentation chunk")
    }
  }
}
}
```

1.2.1 Escaping code

As we will pass code to the `verbfilter` program afterwards, we have to be very careful with some code content that could also be interpreted as LaTeX escape sequences: We have to strip them out:

⟨*escape code*⟩ ≡

```
def codeEscape(code: String): String = {
  code.replace("$","SPEC" + "DOLLAR")
}
```

The problem then is, of course, that we will need to put them back in afterwards.

⟨*escape code*⟩ + ≡

```
def codeUnescape(code: String): String = {
  code.replace("SPEC" + "DOLLAR","\\Dollar")
}
```

1.2.2 Wrapping the document

With the knowledge on how to print blocks, we can go on printing the whole document. If the useHeader flag is set (which it is by default), we generate a standard LaTeX document, the only thing special to note is that we add `scaladefs` which contains macros to format scala output and `scalit` which enables definition indexing.

⟨*print the document*⟩ ≡

```
def writeDoc(out: PrintStream): Unit = {
  if( useHeader ) {
    out.println("\\documentclass[a4paper,12pt]{article}")
    out.println("\\usepackage{amsmath,amssymb}")
    out.println("\\usepackage{graphicx}")
    out.println("\\usepackage{scaladefs}")
    out.println("\\usepackage{scalit}")
    out.println("\\usepackage{fancyhdr}")
    out.println("\\pagestyle{fancy}")
    out.println("\\lhead{\\today}")
    out.println("\\rhead{" + escape(filename) + "}")
    out.println("\\begin{document}")
  }

  blocks zip tangled foreach {
    case ((bs,_),tang) => bs foreach printBlock(out,tang)
  }
  if( useHeader ) {
    out.println("\\end{document}\n\n")
  }
}
```

1.3 Information on the source file

During tangling, we directly interact with the compiler to compile from literate programs. But the compiler can be of much more help - We can for example find out where classes are defined, etc. For this, we reuse the source file class defined for literate compilation[2].

[2] tangle/compilesupport.nw

Another optional parameter is where to find the classes containing the other definitions: This will be used by the compiler to typecheck the code, thus generating the symbols we need.

⟨*Source file information*⟩ ≡

```
import scala.tools.nsc.ast.Trees
class SourceInformation(
  literateFile: tangle.LiterateProgramSourceFile,
  infoClassPath: Option[List[String]]) {
  <Instantiate a compiler>

  <collect information>
  <range of definitions>
}

<Definition info storage>
```

as with the compiler support class, we'll have to instantiate a compiler. However, we will not need to do all the phases, so we overwrite the phases we need:

⟨*Instantiate a compiler*⟩ ≡

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

val settings = new Settings()
infoClassPath match {
  case None ⇒ ()
  case Some(cp) ⇒ settings.classpath.value = cp.head
}

val reporter =
  new ConsoleReporter(settings,null,
                              new java.io.PrintWriter(System.err))
object compiler extends Global(settings, reporter) {
  override protected def builtInPhaseDescriptors:
  List[SubComponent] = List(
    analyzer.namerFactory: SubComponent,
    analyzer.typerFactory: SubComponent
  )
}
```

with the compiler in place, we can now define how to collect the information: Execute the compiler just up to typing and collect them from the syntax tree.

⟨*collect information*⟩ ≡

```
lazy val info = {
  val r = new compiler.Run
  r.compileSources(literateFile :: Nil)

  val typedUnit = r.units.next

  collectDefinitions(typedUnit.body,Map())
}
```

The definition collector needs to have access to the tree case classes. They are part of the compiler. We are very forgiving if, for example we do not find a valid position.

⟨*collect information*⟩ +≡

```
import compiler.{Tree,ClassDef,ModuleDef,PackageDef,
                 DefDef,ValDef,Template}
import DefinitionInfo._
```

We will want to have access to the information on a per-line-basis. However, there will be multiple definitions on one line, so we will need something like a multiset:

⟨*collect information*⟩ +≡

```
type DefMap = Map[Int,Set[Definition]]
```

Another useful state to store is in which class we currently are, so that we can link methods (which might be defined in multiple classes) to a specific class. We overload this method to stay succinct.

⟨*collect information*⟩ +≡

```
def collectDefinitions(t: Tree, acc: DefMap): DefMap =
  collectDefinitions(t,acc,None)

def collectDefinitions(t: Tree,
                       acc: DefMap,
                       container: Option[String]): DefMap = {
```

After these overloaded definitions, let'l begin by getting the source file position:

⟨*collect information*⟩ +≡

```
    val pos = literateFile.positionInUltimateSource(t.pos)
    val line = pos.line match {
      case None ⇒ −1
      case Some(l) ⇒ l
    }
    val before = acc.getOrElse(line,Set())
    t match {
<Handle the class case>
<Handle the object case>
<Handle the package case>
<Handle the method case>
<Handle the value case>
      case other ⇒ acc
    }
  }
```

The accumulator style makes this function rather heavy (note all the folds), but this way we can append the definitions in a predictable style. So, on to the starting point in our tree: The package definition

⟨*Handle the package case*⟩ ≡

```
      case PackageDef(name,stats) ⇒
        (stats foldLeft acc) {
          (defs: DefMap, t: Tree) ⇒ collectDefinitions(t,defs)
        }
```

`defs` holds the current state of the map. Nodes of this package definition will be classes and objects. We will first collect everything in the first class, then pass the definition results to the second class, etc. Here is what we do with classes:

⟨*Handle the class case*⟩ ≡

```
case ClassDef(_,name,tparams,impl) ⇒
  val nameString = name.toChars mkString ""
  val newAcc =
    acc + (line → (before +
                     ClassDefinition(nameString,line,false)))
  impl match {
    case Template(_,_,body) ⇒
      (body foldLeft newAcc) {
        (defs: DefMap, t: Tree) ⇒
          collectDefinitions(t,defs,Some(nameString))
      }
  }
```

Classes have a body which we need to scan, but before we will have to add the class itself to the map. If only that were so easy! Note that there are three basic types of top-level objects on the class level:

- “Normal” classes
- Objects
- Case classes / objects

Note that this is not an either/or decision. A normal way to emulate static members in Scala is the following:

```
class A {
  def aMethod = A.staticOne()
}
object A {
  def staticOne() = ...
}
```

So we will be allowed to have object definitions and class definitions on different lines. The compiler, however, seems to generate both `ClassDef` and `ModuleDef` nodes when we have a case class. Also, the definition map is in an incomplete state, so we can only bet on not seeing another `ClassDef` afterwards by assuming what the compiler generates first the module definition element. This first test tells us to only add contents if we are sure this actually is an object. If we are dealing with a case class, we'll have to indicate this, for this we will use the variable classDefinition.

⟨*Do not add if object is there*⟩ ≡

```
var classDefinition: Option[ClassDefinition] = None
val isRealObject = acc get line match {
  case Some(s) => !(s exists {
    case cd @ ClassDefinition(n,_,_) => {
      if( n == nameString ) {
        classDefinition = Some(cd)
        true
      } else false
    }
    case other => false
  })
  case None =>          true
}
```

If we have already content on this line, then we will not traverse it again, but we will update the fact that we are dealing with a case class.

⟨*Handle the object case*⟩ ≡

```
case ModuleDef(mods,name,impl) ⇒
  val nameString = name.toChars mkString ""
  <Do not add if object is there>
  if(isRealObject) {
    val newAcc = acc + (line →
        (before + ObjectDefinition(nameString,line)))
    impl match {
      case Template(_,_,body) ⇒
        (body foldLeft newAcc) {
          (defs: DefMap, t: Tree) ⇒
            collectDefinitions(t,defs,Some(nameString))
        }
    }
  } else {
    val Some(cd) = classDefinition
    val cc = ClassDefinition(cd.name,cd.l,true)
    acc + (line → (before − cd + cc))
  }
```

For the definitions, We will have to use the same trick as for objects: Some methods are added automatically (like toString, ¡init¿) and should therefore not be included in the index. We solve this by looking up whether a class was defined on the same line. Also, it might be that there exists already a value definition with the same name, in which case we won't add it either:

⟨*Handle the method case*⟩ ≡

```
case DefDef(_,name,tparams,vparams,tpt,_) ⇒
  val isGeneratedMethod = acc get line match {
    case None ⇒ false
    case Some(s) ⇒ s exists {
      case c : ClassDefinition ⇒ true
      case o : ObjectDefinition ⇒ true
      case vd : ValueDefinition ⇒ true
      case _ ⇒ false
    }
  }
  if( isGeneratedMethod ) acc
  else acc + (line →
    (before + MethodDefinition(name.toString,line,container)))
```

We also record value definitions

⟨*Handle the value case*⟩ ≡

```
case ValDef(_,name,_,_) ⇒ {
  val nameString = name.toChars mkString ""
  acc + (line → (before + ValueDefinition(nameString,line,container)))
}
```

One useful command would be to get all the definitions in a range of lines:

⟨*range of definitions*⟩ ≡

```
def getRange(from: Int, to: Int): List[Definition] =
  List.range(from,to + 1) flatMap {
    i ⇒ info.getOrElse(i,Nil)
  }
```

1.3.1 Connection to the weaver

Especially in LaTeXWeaver, we will want to auto-generate some annotations, therefore we'll have to have a value for that:

⟨*compiler help*⟩ ≡

```
import tangle.LiterateProgramSourceFile
val sourcefiles: Option[List[(String,LiterateProgramSourceFile)]] =
  if( useIndex ) {
    Some(tangled map {
      chunks ⇒
        (chunks.filename,
         new tangle.LiterateProgramSourceFile(chunks))
    })
  } else None

val sourceInformation: Map[String,SourceInformation] =
sourcefiles match {
  case None ⇒ Map()
  case Some(sfs) ⇒ Map() ++ (sfs map {
    case (name,sf) ⇒ (name → new SourceInformation(sf,classpath))
  })
}
```

1.3.2 Storing the gathered information

While traversing the tree, we need to store information on what is actually defined. This is particularly:

- Class and trait definitions
- Object definitions
- Method definitions

To each element, we'll want to store the line number so that it can be easily retrieved.

⟨*Definition info storage*⟩ ≡

```
object DefinitionInfo {
  sealed abstract class Definition(line: Int)

  case class ClassDefinition(name: String, l: Int,
                             isCase: Boolean)
    extends Definition(l) {
      override def toString = "" + l + ": Class " + name
    }

  case class ObjectDefinition(name: String,
                              l: Int)
    extends Definition(l) {
      override def toString = "" + l + ": Object " + name
    }

  case class MethodDefinition(name: String,
                              l: Int,
                              container: Option[String])
    extends Definition(l) {
      override def toString = "" + l + ": Method " + name
    }

  case class ValueDefinition(name: String,
                             l: Int,
                             container: Option[String])
    extends Definition(l) {
      override def toString = "" + l + ": Value " + name
    }
}
```

1.3.3 Testing the information gathering

The following command line tool tests the information gathering:

⟨*Source file information*⟩ + ≡

```
object InfoTester {
  def main(args: Array[String]) = {
    import util.LiterateSettings
    import tangle.LiterateProgramSourceFile

    val ls = new LiterateSettings(args)

    val sourceFiles =
      ls.chunkCollections map (
        new LiterateProgramSourceFile(_))

    val infoclasspath = ls.settings get "-classpath"

    val infolist =
      sourceFiles map (new SourceInformation(_,infoclasspath))

    infolist foreach { x => println(x.getRange(0,50)) }
  }
}
```

1.3.4 Adding index information

With the compiler support that we defined before, we will be able to print which functions were defined in a piece of code. The following function adds these bits:

⟨*format index*⟩ ≡

```
  def indexed(content: String, from: Int,
              to: Int, filename: String): String = {
    import DefinitionInfo._
    val ret = new StringBuffer
    ret append content

    sourceInformation get filename match {
      case None => ()
      case Some(si) => {
        val definitions = si.getRange(from,to)
```

The first definitions that we are interested in are which classes are defined:

⟨*format index*⟩ + ≡

```
val classes = definitions filter {
    case cd: ClassDefinition => true
    case _ => false
}
```

Then methods. Here, we want to filter out the generated methods

⟨*format index*⟩ +≡

```
val methods = definitions filter {
    case MethodDefinition(name,_,_) => true
    case _ => false
}
```

Values and objects are filtered in a same way

⟨*format index*⟩ +≡

```
val values = definitions filter {
    case v : ValueDefinition => true
    case _ => false
}
val objects = definitions filter {
    case o: ObjectDefinition => true
    case _ => false
}
```

Now for the LaTeX output generated: We use the commands defined in `scalit.sty` so that we can change presentation later on. Note that no real output needs to be generated.

⟨*format index*⟩ +≡

```
            if (!classes.isEmpty || !methods.isEmpty ||
                  !objects.isEmpty) {
<output class definition info>
<output object definition info>
<output method definition info>
<output value definition info>
            }
            ret append "\n\n"
          }
        }

        ret.toString
      }
```

First the classes. The command `classdefinition` takes care of them. At the moment, we are not indicating case classes specially.

⟨*output class definition info*⟩ ≡

```
classes foreach {
  case ClassDefinition(name,_,caseClass) => {
    ret append "\\classdefinition{" + name + "}\n"
  }
  case _ => ()
}
```

Second come the object definitions.

⟨*output object definition info*⟩ ≡

```
objects foreach {
  case ObjectDefinition(name,_) => {
    ret append "\\objectdefinition{" + name + "}\n"
  }
  case _ => ()
}
```

Methods are prefixed by the class in which they are defined. Note the flaw in this system: This way we are only recording one level of classes, thus generating a flat index. Also, the body of the methods is not further traversed, so inner functions are not detected.

⟨*output method definition info*⟩ ≡

```
methods foreach {
  case MethodDefinition(name,_,cont) => {
    ret append "\\methoddefinition{"
    cont match {
      case None => ()
        case Some(n) => ret append escape(n)
    }
    ret append "}{"
    ret append escape(name) + "}\n"
  }
  case _ => ()
}
```

Value definitions are also noted. This might be a bit overkill, so in a further stage we could filter for only publicly accessible values.

⟨*output value definition info*⟩ ≡

```
values foreach {
  case ValueDefinition(name,_,cont) => {
    ret append "\\valuedefinition{"
    cont match {
      case None => ()
        case Some(n) => ret append escape(n)
    }
    ret append "}{"
    ret append name + "}\n"
  }
  case _ => ()
}
```

1.4 The command line application

The command line application gets quite a bit more complicated than before: Not only do we scan the code blocks but we also tangle the source! This way we can assure that we do not reference code blocks in the text that were not defined. Also, in a later stage we could use compiler information to see what is defined where etc. `LiterateSettings`[3] lets us read in multiple files to form one LaTeX document.

Specifically, it takes the following options:

-vf t/f if t, then apply verbfilter on source. Default true

-idx t/f if t, then generate definition index. Default false

-classpath path Classpath to be used for reference to other classes if index is built.

⟨*The command line application*⟩ ≡

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

    val ls = new LiterateSettings(args)

    val blocks : List[(Stream[markup.Block],String)] = ls.blocks
```

This is the same as with tangle: We want to extract code blocks from the line format. We will actually partially execute the tangle phase: The part where we put chunks together.

But first, let us deal with the title of the file. If one is specified on the command line, we use this. Otherwise, we use the name of the first file given as input.

⟨*The command line application*⟩ +≡

```
    val filename = ls.settings get "-title" match {
      case Some(x::xs) => x
      case _ => blocks match {
        case (_,name) :: xs => name
        case Nil => ""
      }
    }
```

[3]See util/commandline.nw

Some settings are taken out of the settings object directly, where to send output is handled specially.

⟨*The command line application*⟩ + ≡

```
val classpath = ls.settings get "-classpath"

val verbfilter = ls.settings get "-vf" match {
  case Some(x :: xs) => x(0) == 't'
  case _ => true
}

val index = ls.settings get "-idx" match {
  case Some(x :: xs) => x(0) == 't'
  case _ => false
}
```

This follows a very general pattern: If the option does not exist, it gets a default value, otherwise it depends on the option whether we interpret it as truth value or otherwise. The following option is on whether to include a header in the generated `tex` file. By default, we print one:

⟨*The command line application*⟩ + ≡

```
    val header = ls.settings get "-header" match {
      case Some(x :: xs) => x(0) == 't'
      case _ => true
    }

    val weaved = LatexWeaver(blocks,ls.chunkCollections,
                             verbfilter,index,classpath,filename,
                             header)

    weaved.writeDoc(ls.output)
  }
}
```

Definitions