AmyDoc

Computer Language Processing '18 Final Report

Simon Perriard Hédi Sassi
EPFL
simon.perriard@epfl.ch, hedi.sassi@epfl.ch

1. Introduction

The compiler we implemented in the labs allows us to compile Amy code into web assembly. The compiler is built using a pipeline of 6 stages:

- Lexer :

  Takes input files and outputs a stream of tokens representing elements of code (keywords, identifiers, syntax elements etc..).

- Parser

  Takes the stream of tokens and constructs and outputs an AST using an LL1 grammar that corresponds to Amy specifications.

- NameAnalyser

  Takes the AST and checks that names and identifiers are not conflicting with respect to their scope (static scoping) and outputs the AST with unique identifiers instead of names and a symbol table that contains all functions and class definitions.

- TypeChecker

  Takes the AST and the symbol tables and checks that the program is correct with respect to types. If it typechecks, the AST and the symbol table are both forwarded to the next stage.

- CodeGen

  Takes the AST and the symbol table and constructs the web assembly code that is wrapped in a "Module" object and passed on to the final stage.

- CodePrinter

  Takes the web assembly wrapped in the "Module" and writes all the web assembly files in a folder. This stage is the end of the pipeline.

The compiler is built according to the Amy specification and, thus supports single line and multi-lines comments.

Although comments are necessary for code comprehension, they are not convenient when one has to work within a team or when writing APIs. In those cases, documentation is necessary in the way to avoid spending hours looking at all the comments written in the different modules of an API.

Our extension aims to fill that gap by creating a compiler-embedded documentation tool for Amy code. This tool will allow programmers to generate HTML and markdown files for each module just by compiling their code. In that way, the documentation can easily be published on a website or on a Github repository.

2. Examples

First, let's see how to use AmyDoc. As it is also the case in JavaDoc we have delimiters to specify that it is actually documentation, we chose our delimiter to be '~' (tilde).

We also implemented the @param, @return and @see tags, the two first work like the JavaDoc's ones and are used to specify respectively what the parameter should be and what the function returns.Now the @see tag allows the programmer to build an hyperlink to another module or function's doc.

Note that the documentation generator will check that the name given to @param actually exists in the parameters of the function's definition, and @see will check for the existence of the specified module and function in the module. If an error is detected, the doc generation will fail at that point.

As the intermediate output is in Markdown, it is possible to use embedded (**text** for bold, _text_ for

italic, ...) Markdown or any HTML supported embedding in Markdown directly within the doc.

The first example is a working one :

```
object O{
  abstract class Option
  case class None() extends Option
  case class Some(v: Int) extends Option
}

object L{

  abstract class List
  case class Nil() extends List
  case class Cons(h: Int, t: List) extends List

  ~
  This function returns the head of the list _if any_
  @param l a list
  @return an option (@see O) with the head
  of the list if any, the get method (@see O.isDefined)
  will be needed to check whether the option contains
  something
  ~
  def headOption(l: List): O.Option = {
    l match {
      case Cons(h, _) => O.Some(h)
      case Nil() => O.None()
    }
  }
}
```

Here _if any_ will be output as *if any* and @see O will become a hyperlink to the doc containing the O module. The final HTML output for the headOption doc will look like the following snippet :

headOption(l: List): Option

This function returns the head of the list *if any*
Parameter : l a list
Return : an option (see O) containing the head of the list if any, the get method (see O.isDefined) will be needed to check whether the option contains something

The second example contains errors that will be detected during the compilation :

```
object O{
  abstract class Option
  case class None() extends Option
  case class Some(v: Int) extends Option
}

object L{

  abstract class List
  case class Nil() extends List
  case class Cons(h: Int, t: List) extends List

  ~
  This function returns the head of the list **if any**
  @param li a list
  @return an option (@see Q) with the head
  of the list if any
  ~
  def headOption(l: List): O.Option = {
    l match {
      case Cons(h, _) => O.Some(h)
      case Nil() => O.None()
    }
  }
}
```

Here the compilation will fail at '@param li' because there is no such parameter in the headOption function definition. In the case the parameter's name was correct, the compilation would still fail because the module Q (@see Q) does not exist.

3. Implementation

The first step for building the documentation tool was to decide where to insert it in the pipeline. Since we wanted the documentation to be generated after each *successful* compilation, we decided to place the DocGen stage just between the TypeChecker and CodeGen. Doing so, we still have access to the symbol table with all information necessary to perform checks on the documentation.

Then, we modified the Lexer to generate a new kind of token: "DOC(String)" that is very similar to the "StringLit(String)" token and that will wrap the documentation.
Since we added a new token, we had to add it in the grammar by allowing an optional documentation just before a function definition:

'FunDef ::= 'OptDoc ~ DEF() ~ 'Id ...,
'OptDoc ::= DOCSENT | epsilon(),

In the AST, we chose to add an Option[String], representing the documentation, to the function definition.

To do it we added the 'doc: Option[String]' parameter to FunDef definition in TreeModule and adapted the code accordingly. Thus, in ASTConstructorLL1, we had to modify the way we construct the AST to take into account the optional documentation by overriding the constructDef0 function.

Finally, we implemented the DocGen stage. We iterate on each module and parse the documentation for each function in the module. We check that parameters names and links are valid using the symbol table and then, we generate Markdown files for each module.
The Mardown files can later be converted to html using *pandoc* or used as it is in, for example, github repositories.

4. Possible Extensions

AmyDoc is function-centered since it only allows to write documentation for functions and Markdown files are generated by Modules. It could be interesting to allow programmers to write documentation for a whole class hierarchy. This ways we could also generate a more class-oriented documentation (as in Javadoc) with class hierarchy schemes and other features, that helps programmers to keep the general overview of a project.
Another extension could be the possibility for a programmer to leave its details in the documentation via a special annotation (e.g. @author). This would allow the creation of a contact list in case someone need to ask a question about a specific part of the code.