Tastydoc: a documentation tool for dotty using Tasty files

Bryan Abate

Supervised by Aleksander Boruch-Gruszecki

May 28, 2019

Abstract

The current documentation tool (Dottydoc) relies on compiler internals and low level code. The tool introduced here aims to build a program not dependant on compiler internals but instead use Tasty files which are output when a Scala program is compiled.

It also aims at providing a tool with less bugs, more features and which is more easily maintenable.

For flexibility the output is in Markdown instead of the commonly used HTML.

Contents

Chapter 1

Introduction

A documentation tool is a program generating information (often in the form of HTML files) about projects, it usually includes: method signatures, user written documentation, link to other pages, etc. In Dotty, the current tool for generating Scala project documentation is called Dottydoc. The tool introduced here, called Tastydoc, aims to improve on Dottydoc and replace it.

Tastydoc makes use of TASTy files, they are output when a program is compiled, they contain information about the source code of a program. Tastydoc consumes such files to extract information about the code, that way it is completely independant from the compiler and can be used on its own. The tool tries to be as close as possible to the structure proposed in Dottydoc so that it can reuses partial portion of its code with minor modificaitions. As though documentation is often output as HTML files, here we choose to output Markdown files for reasons we detail further in the report.

The tool was developped with a few goals in mind. It should be independant from the compiler, hence the use of TASTy files. It should produce humanly consumable files, hence the choice of Markdown as an output. It should have as much features as Dottydoc and produce a code easily maintenable.

The report is organized in the following structure: First we will talk about the features of the tool, then give a high-level and a low-level architectural description. Dottydoc and Tastydoc will then be compared. We will finally talk about problems encountered during the development, further work and sumarise the project.

Chapter 2

Features

We will list here the most important features of Tastydoc. The tool has knowledge of the full signature of entities (classes, defs, vals, etc.), of classes/traits/objects/packages members, of classes constructors, known subclasses of a trait or class, companion and of user written documentation (including @ notations). It offers an easy to use interface (called Representation, see) to access all these information for anyone willing to write their own documentation tool without worrying to consume TASTy files. The tool relies on TASTy files and is hence independant from the compiler.

The intermediate structure is similar do Dottdoc `Entity`[1] which allows for an easy reuse of part of Dottydoc code with minor modifications. Right now, the tool uses some code from Dottydoc for parsing user documentation. But we can think of using it to produce the same HTMl files as does Dottydoc but without relying on compiler internals for example.

Both syntax of user documentation is supported, that is the Wiki and the Markdown syntax. Linking to entities (classes, methods, etc.) is possible inside the user documentation.

Tastdoc is able to link to type defined outside of the current TASTy files (as well as the one inside the file). This is particularly useful for linking to parameters and return types, companions and parents.

The tool produces Markdown documentation files which are easy to modify and read.

[1] `dotty/tools/dottydoc/Entity.scala`

Chapter 3

Architecture

3.1 High-level architectural descriptions

3.1.1 TASTy

TASTy is a data format aimed at serializing compiler information about code. Dotty produces these files for each classes, objects without companion, traits and packages if it contains something else than classes, traits or objects. From its nature it should be perfect to extract information from. This is done through the use of the reflect API and making a class extanding `TastyConsumer`[1].

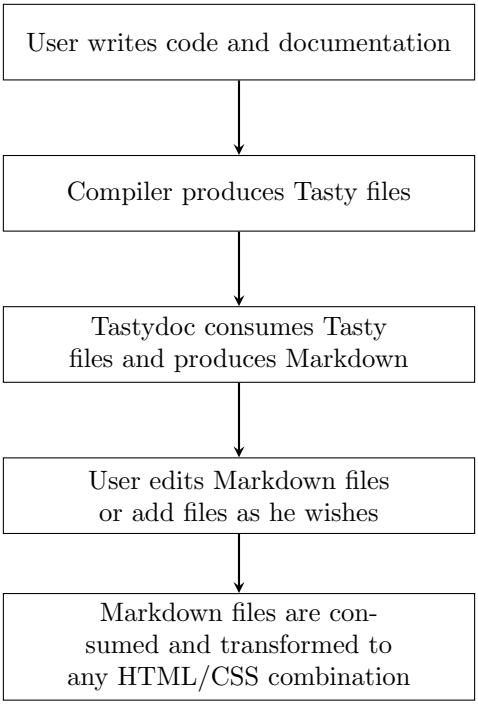
3.1.2 Markdown

Outputing Markdown instead of HTML has several advantages. It is easy to edit by hand and easily previewed. Users can easily add their own text or own file. A user could, for example, write a Markdown file and link it in the documentation of class or do the opposite, link a documentation page in a Markdown file.

Markdown files can be easily uploaded to GitHub or GitLab, hence it would be extremly easy to upload documentation files along the code of a project.

In addition to that lots of tools are available to consume Markdown files and output other format such as HTML or PDF. Making of Markdown a nice intermediate representation. From this we can imagine a pipeline like this:

[1] `scala/tasty/file/TastyConsumer.scala`



3.1.3 Reuse Dottydoc's intermediate representation

???Said enough in Features??? Allows reusing Dottydoc's output; Dottydoc already has a complex templating mechanism which allows attaching blogposts to documentation;

3.2 Low-level architecture

3.2.1 TastyExtractor

File: `dotty/tastydoc/TastyExtractor.scala`

A trait containing useful methods for extracting information from the reflect API.

3.2.2 References

File: `dotty/tastydoc/references.scala`

Object containing case classes. References are classes containing information about a specific type to be able to link to it later on. Inspired from Dottydoc.

3.2.3 TastyTypeConverter

File: `dotty/tastydoc/TastyTypeConverter.scala`

Trait containing methods for converting from Reflect types to References.

3.2.4 Representation

File: `dotty/tastydoc/representations.scala`

Implements both `TastyExtractor` and `TastyTypeConverter`

A Representation contains all the information of a specific entity. The logic is as follows: different trait such as `Modifiers` or `Members` and classes which implement those trait. This logic is inspired by dotty-doc entities (`dotty/tools/dottydoc/Entity.scala`).

A Representation take a reflect type such as `reflect.ClassDef` and extract every information from it using mostly `TastyExtractor` functions.

Representation can then be easily used for printing, their content is self explanatory and no knowledge of Tasty is required to use them as the implementation is not exposed from the outside.

We give below the existing Representation, a quick description and their signature (without parameters).

- `PackageRepresentation` For packages

  ```
  class PackageRepresentation(...)
  extends Representation with Members
  ```

- `ImportRepresentation` For import, never used in the tool

  ```
  class ImportRepresentation(...)
  extends Representation
  ```

- `ClassRepresentation` For class, object and trait, including case class and case object.

  ```
  class ClassRepresentation(...)
  extends Representation with Members
  with Parents with Modifiers with Companion
  with Constructors with TypeParams
  ```

- `DefRepresentation` For definitions

  ```
  class DefRepresentation(...)
  extends Representation with Modifiers
  with TypeParams with MultipleParamList
  with ReturnValue
  ```

- `ValRepresentation` For val and var

  ```
  class ValRepresentation(...)
  extends Representation with Modifiers
  with ReturnValue
  ```

- `TypeRepresentation` For type alias

  ```
  class TypeRepresentation(...)
  extends Representation with Modifiers
  with TypeParams
  ```

- `EmulatedPackageRepresentation` This Representation is a trick for regrouping every PackageRepresentation of a same package under the same Representation. Indeed the way Tasty works is that for each class or trait it will have a Tasty file and the top of `reflect.Tree` is a`reflect.PackageClause`, hence when converting multiple classes of the same package, there will be mutiple `PackageRepresentation` of the same package. In the point of view of usability, this is not the best and this Representation is here to counter this problem. When using an `EmulatedPackageRepresentation`, for the user it looks like he is using a `PackageRepresentation` containing all the members of the package.

3.2.5 DocPrinter

File: `dotty/tastydoc/DocPrinter.scala`

Object with methods for formatting Representations and References and writing them to files. Basically handle all the formatting and printing logic of the tool.

3.2.6 mdscala

File: `dotty/tastydoc/mdscala.scala`

3.2.7 TastyDocConsumer

File: `dotty/tastydoc/TastyDocConsumer.scala`

Extends TastyConsumer and consumes Tasty Files to produce Representations.

3.2.8 Main

File: `dotty/tastydoc/Main.scala`

Manages the workflow.

Command line arguments are:

- **-syntax** {*wiki* or *markdown*} Syntax to use for user documentation parsing
- **-packagestolink** {*regex1 regex2 ...*} Regexes to specify which packages should be linked when formatting Reference
- **-classpath** {*URI*} Extra classpath for input files
- **-i** {*file1 file2 ...*} Tasty files

3.2.9 mdscala

File: `dotty/tastydoc/mdscala.scala`

Object with methods for outputing markdown (do not handle escaping).

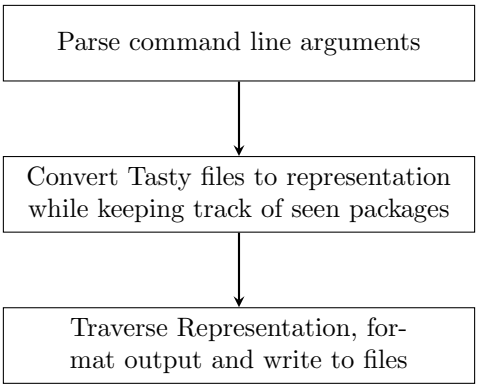
3.2.10 User documentation parsing

Files are in the package `dotty.tastydoc.comment`.

See section 12.4

3.3 Workflow

Main workflow is as follows:



Chapter 4

Comparison between DottyDoc and TastyDoc

*** Output/feature differences

Chapter 5

Problems

*** No good Markdown library

Chapter 6

Remaining work

*** Highlighting type lambdas *** ...

Chapter 7

Summary

Chapter 8

BELOW IS OLD REPORT, STOP HERE

Chapter 9

Introduction

In dotty the documentation generation tool is called `dotty-doc`. However the tool is flawed in many ways, these flaws include:

- Reliance on compiler internals
- Low level and unecessarily complex code which make it hard to maintain
- Not maintened and not documented
- Not flexible in its output, it outputs HTML/css in a hard to modify way
- Malformed type output such as `[32m"getOffset"[0m`
- Classes often show to be extending Object instead of their superclass. Example: `https://dotty.epfl.ch/api/scala/Conversion.HTML`

  abstract class Conversion [–T, +U] **extends** Object **with** Function1
- Annotations which sould not be display, sometimes are. Example:
- Lacks some features, especially:
  - No known subclasses
  –

The tool introduced here aims to address all these issues while providing with a code easy to maintain and easy to adapt to every need.

One major difference with currrent documentation tools is that the output is in Markdown instead of HTML, this has some pros and cons which will be discussed below.

Althought dotty-doc has some problems, the current tool will draw inspiration for some of its architecture which will allow for its parsing code for user documentation (modified to handle this structure and output Markdown) to be reused.

The report will follow the structure described in this introduction and will conclude with problems of the current implementation and further work.

Chapter 10

Features

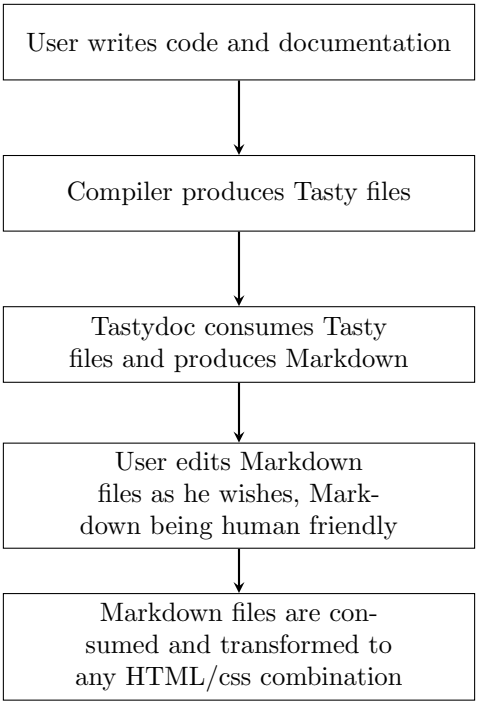
Chapter 11

Output format

Before describing the output, we list the reasons, pros and cons of using a Markdown output.

11.1 Reasoning behind Markdown

The major reason to use Markdown instead of HTML is to think of the documentation tool as a pipeline which would look like this:



With such a pipeline we do not limit the user to use a specific output for the documentation, he can easily remove, add parts, etc. manually and then use an HTML format of his choosing as Markdown only specify the structure not how it should be displayed.

11.2 Pros and Cons

Using Markdown as an intermediate output format comes with some pros and cons comparing to an HTML output:

- **Pros:**
  - Human readable and editable, meaning part of the documentation can be written manually
  - Easy to infer the format of the output to extend it manually
  - Live preview and editing
  - Only define the structure of the output not how to display it
  - Easily pipelined to another format (HTML, reStructuredText, etc.)
- **Cons:**
  - No Scala (or Java) library available to output markdown with escaping
  - Markdown fenced code block cannot contain links which forces us to use html code block, hence loose syntax highlight in Markdown and not output "pure" Markdown.
  - Markdown does not have "division" like `<div>` in HTML to better structure the output

11.3 Output structure

In the following section, we describe how the markdown output is structured.

11.3.1 class, object and trait

Have their own `.md` file.

1. Name (header 1)
2. Companion object (header 2)
3. Signature (HTML pre + code)
4. User documentation
5. Annotations (header 2)

6. Known subclasses (header 2)
7. Constructors (header 2)
   (a) Name + paremeters (HTML pre + code)
   (b) User documentation
8. Members (header 2) (In order: Abstract Type Members, Concrete Type Members, Abstract Value Members, Concrete Value Members)
   (a) Follows structure described in subsection 11.3.3

11.3.2 package

Has its own `.md` file.

1. Name (header 1)
2. Members (header 2)
   (a) Follows structure described in subsection 11.3.3 and subsection 11.3.4

11.3.3 type alias, class (simplified format), def, val and var

No `.md` file.

1. Name (header 3)
2. Annotations + signature (HTML pre + code)
3. User documentation

11.3.4 package (simplified format)

No `.md` file.

1. name + link (HTML pre + code)

Chapter 12

Architecture

12.1 General architecture

12.1.1 TastyExtractor

File: `dotty/tastydoc/TastyExtractor.scala`

A trait containing useful methods for extracting information from the reflect API. Methods are:

- `extractPath` extract the extractPath
- `extractModifiers` extract all the useful modifiers, including scope modifiers
- `extractComments` extract user documentation, more speficially a function requiring a map of packages needed for parsing
- `extractClassMembers` extract the members of a class, including the inherited ones
- `extractParents` extract the parents of a class, object or trait
- `extractKind` extract information (is it an object? a trait? a class? a case?) on a `reflect.ClassDef`
- `extractCompanion` extract the companion object or class if it exists
- `extractAnnotations` extract the annotations
- `extractPackageNameAndPath` extract the name and the path from a `pid`

12.1.2 References

File: `dotty/tastydoc/references.scala`

Object containing case classes. References are classes containing information about a specific type to be able to link to it later on. Inspired from Dottydoc.

12.1.3 TastyTypeConverter

File: `dotty/tastydoc/TastyTypeConverter.scala`

Trait containing methods for converting from Reflect types to References.

12.1.4 Representation

File: `dotty/tastydoc/representations.scala`

Implements both `TastyExtractor` and `TastyTypeConverter`

A Representation contains all the information of a specific entity. The logic is as follows: different trait such as `Modifiers` or `Members` and classes which implement those trait. This logic is inspired by dotty-doc entities (`dotty/tools/dottydoc/Entity.scala`).

A Representation take a reflect type such as `reflect.ClassDef` and extract every information from it using mostly `TastyExtractor` functions.

Representation can then be easily used for printing, their content is self explanatory and no knowledge of Tasty is required to use them as the implementation is not exposed from the outside.

We give below the existing Representation, a quick description and their signature (without parameters).

- `PackageRepresentation` For packages

  ```
  class PackageRepresentation(...)
  extends Representation with Members
  ```

- `ImportRepresentation` For import, never used in the tool

  ```
  class ImportRepresentation(...)
  extends Representation
  ```

- `ClassRepresentation` For class, object and trait, including case class and case object.

  ```
  class ClassRepresentation(...)
  extends Representation with Members
  with Parents with Modifiers with Companion
  with Constructors with TypeParams
  ```

- `DefRepresentation` For definitions

  ```
  class DefRepresentation(...)
  extends Representation with Modifiers
  with TypeParams with MultipleParamList
  with ReturnValue
  ```

- `ValRepresentation` For val and var

  ```
  class ValRepresentation(...)
  extends Representation with Modifiers
  with ReturnValue
  ```

- `TypeRepresentation` For type alias

  ```
  class TypeRepresentation(...)
  extends Representation with Modifiers
  with TypeParams
  ```

- `EmulatedPackageRepresentation` This Representation is a trick for regrouping every PackageRepresentation of a same package under the same Representation. Indeed the way Tasty works is that for each class or trait it will have a Tasty file and the top of `reflect.Tree` is a `reflect.PackageClause`, hence when converting multiple classes of the same package, there will be mutiple `PackageRepresentation` of the same package. In the point of view of usability, this is not the best and this Representation is here to counter this problem. When using an `EmulatedPackageRepresentation`, for the user it looks like he is using a `PackageRepresentation`.

12.1.5 DocPrinter

File: `dotty/tastydoc/DocPrinter.scala`

Object with methods for formatting Representations and References and writing them to files. Basically handle all the formatting and printing logic of the tool.

12.1.6 mdscala

File: `dotty/tastydoc/mdscala.scala`

12.1.7 TastyDocConsumer

File: `dotty/tastydoc/TastyDocConsumer.scala`

Extends TastyConsumer and consumes Tasty Files to produce Representations.

12.1.8 Main

File: `dotty/tastydoc/Main.scala`

Manages the workflow.

Command line arguments are:

- **-syntax** {*wiki* or *markdown*} Syntax to use for user documentation parsing
- **-packagestolink** {*regex1 regex2 ...*} Regexes to specify which packages should be linked when formatting Reference
- **-classpath** {*URI*} Extra classpath for input files
- **-i** {*file1 file2 ...*} Tasty files

12.1.9 mdscala

File: `dotty/tastydoc/mdscala.scala`

Object with methods for outputing markdown (do not handle escaping).

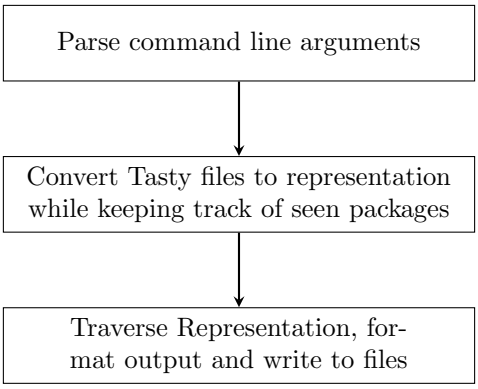
12.1.10 User documentation parsing

Files are in the package `dotty.tastydoc.comment`.

See section 12.4

12.2 Workflow

Main workflow is as follows:



12.3 Formatting and writing to files

Useful ?

12.4 Use of dotty-doc for parsing

Part of dotty-doc code is a parser for user documentation, basically parsing it and also getting every @. Althought the code is more complex than it should, it does the job right and hence is used for this tool. Obviously it needed some adjustements but as we have a very similar structure as Dott-doc's entities, the code nearly worked out of the box when refactoring `Entity` to `Representation`.

Note that it is using a library called flexmark-java[1] for parsing Markdown.

The parser handles the wiki syntax as well as the Markdown syntax for user documentation.

[1] `https://github.com/vsch/flexmark-java`

Chapter 13

Problems encountered and further work

13.1 Problems encountered

Here is a listing of a few problems encountered:

- All the cons listed in section 11.2
- Multi level list are not parsed correctly when ordered and unordered list are combined, they are parsed as 2 separate list. This is a problem in flexmark-java
- If a type (class or type alias) inside a class has the same name as a def/-val/var in the same class, for linking to them directly (anchor basically), there is no way to do it in markdown (there is no id, except for title). To counter this problem we list types before values so that it will always jump to the first one

13.2 Further work

Further work would include:

- Make a library for outputing markdown with escaping. For example, right now, if a method is named `#` it will cause a problem
- Handle correctly Type Lambdas, right now they are handeled as constant reference
- Propose a default HTML/CSS template for people who just want something working out of the box.
- Simplify user documentation parsing code taken from Dotty-doc

Chapter 14

Conclusion