# Tastydoc: a documentation tool for dotty using Tasty files

Bryan Abate

Supervised by Aleksander Boruch-Gruszecki

May 29, 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].

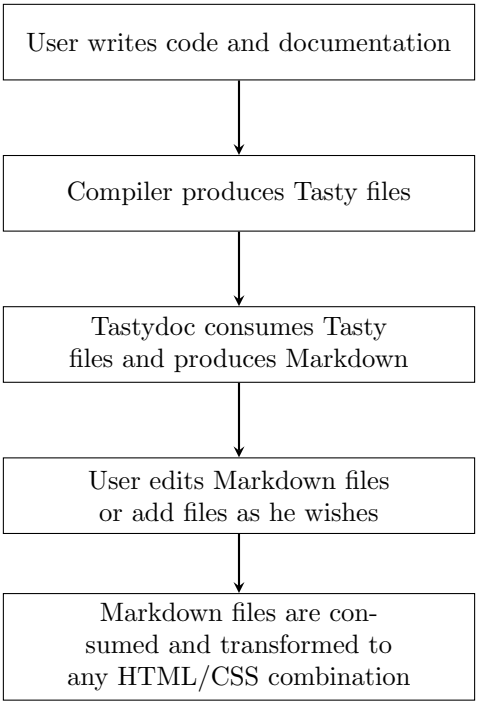
### 3.1.2 Markdown

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. This is inspired from Dottydoc. There exists a reference for all the existing special type such as "or type". Referencing a user defined type (such as a class or a type alias) is done using a label (the name of the type, like "String") and a path (the full path in which it is defined, like "/scala/Predef/).

### 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`

Object implementing both `TastyExtractor` and `TastyTypeConverter`

A Representation contains all the information of a specific entity. The logic is as follows: there exist different trait such as `Modifiers` or `Members` and classes which implement those trait. This logic is inspired by dotty-doc Entities[2].

A Representation take a reflect class such as `reflect.ClassDef` and extract every information from it using mostly `TastyExtractor` functions while converting types to References using `TastyTypeConverter` 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

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

```
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 to Markdown 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`

Object with helper methods for outputing markdown. Unfortunately it does not handle escaping as it is a non trivial task and would have required a significant amount of time.

### 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 (see section 3.3).

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 User documentation parsing

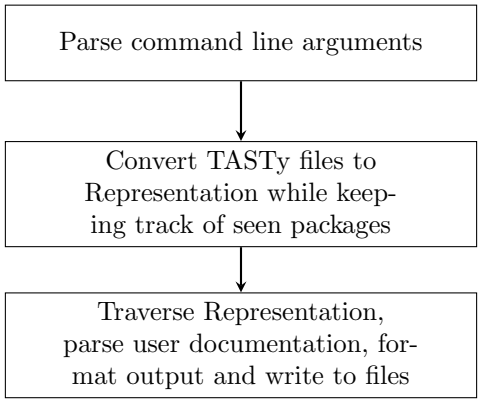
Files are in the package `dotty.tastydoc.comment`.

These files are taken from Dottydoc (slightly modified to work with this architecture and output Markdown instead of HTML) and are responsible for parsing user written documentation.

Note that it is using a library called flexmark-java[3] for parsing user documentation written in Markdown.

## 3.3 Workflow

Main workflow is as follows:




As we can see, parsing user documentation is only done after converting to Representation. Indeed to be able to link methods, classes, etc. from user documentation we need to have converted everything to Representation first otherwise there is no way to know what is linked (a method? a class? an object?).

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

# Chapter 4

# Comparison between Dottydoc and Tastydoc

Here we compare Dottydoc and Tastydoc in term of features, bugs and output.

Dottydoc heavily relies on compiler internals and hence cannot be detached from it. Tastdoc relies on TASTy files and hence can be completely separate from the compiler and can be distributed as a separate tool.

Dottydoc relies on low level and complex code where Tastydoc relies on the comprehensive Reflect API while trying to be as clear and as maintenable as possible.

Dottydoc outputs HTML/CSS with a preformatted template and offers no way to customize anything. Also HTML/CSS forces the user to host it on a website where Markdown can simply be viewed on an online git repository without further tasks. Tastydoc counters this problem by outputting Markdown.

Dottydoc is not maintened and hence does not follow the evolution in Master. This opens to bugs for example it does not take care of colors in the `show` method and it results in faulty type or method output like `[31m2L[0m`[1]. Classes often show to be extending Object instead of their superclass because Object is given first when asking for the parents list to the compiler, example[2]:

```
abstract class Conversion [ -T, +U ]
extends Object with Function1
```

Annotations which should not be displayed (like `@child`) are displayed and sometimes manye time[3].

Some features are present in Tastydoc but not in Dottydoc. Tastydoc keeps track of all seen subclasses of a class and link to them.

[1] `https://dotty.epfl.ch/api/dotty/runtime/LazyVals$.html`
[2] `https://dotty.epfl.ch/api/scala/Conversion.HTML`
[3] `TODO`

# Chapter 5

# Problems

A few problems were encountered during the development, some could not be fixed for reasons detailled below.

The output being in Markdown we need a library to take a String as input and output Markdown, like "format this input to a header of level 1". The library also needs to escape the input, indeed if we have a String "#", if we do not escape it, Markdown will interpret it as a header of level 1 and this is not the expected behavior. This is where we have a problem because no such library exists in Java or Scala.

We use code blocks for formatting signatures and indicating we have a piece of code. In these code blocks we also want to be able to make links pointing to the documentation page of a type. Unfortunately Markdown fenced code blocks do not allow to have link in them. To counter this problem we use HTML pre and code tag which are also understood by Markdown viewer but this causes two problems. First it is not pure Markdown anymore and then we loose the language syntax highlight from Markdown fenced code blocks.

Markdown does not have "division" with classes like `<div>` in HTML which could be useful to better structure the page into sections.

Most Markdown viewer do not understand user specified id which could be useful for linking to specific things in the same page. Like for example linking to type parameters. Hence we do not use custom ids for linking in this project and only use automatically generated ids from headers.

If we have twice the same header name, when linking to this header it is ambiguous which header to link to. To mitigate this problem, we first output types and then values so that when linking it links to the first one appearing in the page which would be a type and it is what we want in most cases.

Flexmark-java Markdown parser does not understand when ordered and unordered lists are combined, they are parsed as 2 separate list instead of a multi-level list.

# Chapter 6

# Remaining work

Althought the project is nearly features complete. Some work needs to be done to perfect the tool.

As seen in chapter 5 we do not escape anything before outputting Markdown this is a problem that needs to be adressed by making a library capable of escaping Markdown. This is not a trivial task as Markdown has a lot of different available syntaxes, especially when we extends Markdown with some features like tables.

Type Lambdas are complex types to handle and convert to References, right now they are converted to constant References. This is not the expected behavior and the output is not the one we expect and no linking can be done when using constant Reference. It is however acceptable for a first version as type lambdas do not appears that often.

Markdown files for documenation are good for all the reasons mentionned in this report but some people still want a good HTML/CSS documentation. The aim here would be to either take back what Dottydoc has and use it with our code, either build something new. The first case should be pretty straightforward as our intermediate representation looks very much alike Dottydocs. The biggest change would be for References as links are handeled a bit differently in this tool.

# Chapter 7

# Summary