Tastydoc: a cocumentation tool for dotty using Tasty files

Bryan Abate

Supervised by Aleksander Boruch-Gruszecki

May 23, 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

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 2

Features

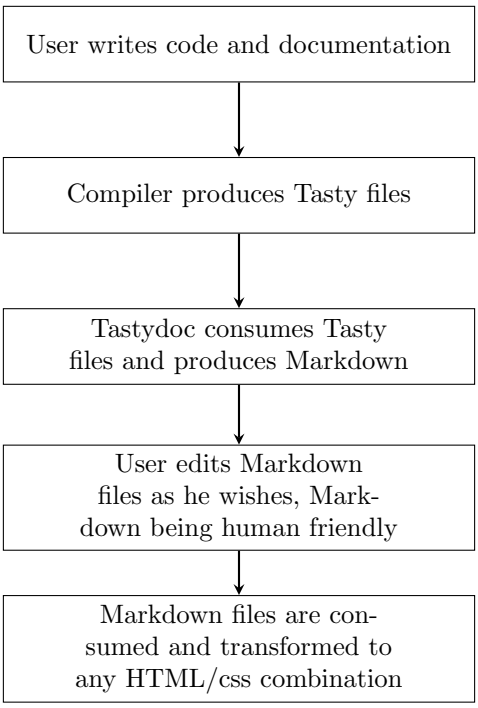
Chapter 3

Output format

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

3.1 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.

3.2 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

3.3 Output structure

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

3.3.1 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 3.3.3

3.3.2 package

Has its own `.md` file.

1. Name (header 1)
2. Members (header 2)
   (a) Follows structure described in subsection 3.3.3 and subsection 3.3.4

3.3.3 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

3.3.4 package (simplified format)

No `.md` file.

1. name + link (HTML pre + code)

Chapter 4

Architecture

4.1 General architecture

4.1.1 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`

4.1.2 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.

4.1.3 TastyTypeConverter

File: `dotty/tastydoc/TastyTypeConverter.scala`

Trait containing methods for converting from Reflect types to References.

4.1.4 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.

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`.

4.1.5 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.

4.1.6 mdscala

File: `dotty/tastydoc/mdscala.scala`

4.1.7 TastyDocConsumer

File: `dotty/tastydoc/TastyDocConsumer.scala`

Extends TastyConsumer and consume Tasty Files.

4.1.8 Main

File: `dotty/tastydoc/Main.scala`

Manages the workflow.

4.1.9 mdscala

File: `dotty/tastydoc/mdscala.scala`

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

4.1.10 User documentation parsing

See section 4.4

4.2 Workflow

4.3 Formatting and writing to files

4.4 Use of dotty-doc for parsing

Chapter 5

Problems encountered and further work

5.1 Problems encountered

Here is a listing of a few problems encountered:

- All the cons listed in section 3.2
- 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
- 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

5.2 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

Chapter 6

Conclusion