ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Tooling support for Dotty Enum

Samuel Chassot

School of Computer and Communication Sciences
Semester Project Report

Supervisor
Allan Renucci
EPFL / LAMP

Supervisor
Prof. Martin Odersky
EPFL / LAMP

January 2019

Tooling support for Dotty Enum

Samuel Chassot
EPFL, Switzerland
samuel.chassot@epfl.ch

Abstract

The Dotty compiler introduced support for enumerations. To improve tooling support for enumerations, we updated the library Scalameta to make it handle them. Then we updated another library that uses Scalameta to format Scala code, Scalafmt. Finally, we developed a small extension for Visual Studio Code that handles the Document Formatting Request and uses Scalafmt to format Scala source code directly from the editor.

1 Introduction

The goal of this project was to improve tooling support for Dotty enumerations. We chose Scalameta as a building block to achieve this goal.

Dotty is the project name that englobes all the technologies that are considered to be included in the version 3 of Scala.

Dotty introduced the enumerations. They are very similar to a class except they can contain values. These values can have simple or more evolved forms. Let us look at an example to illustrate.

```
enum Color(val rgb: Int){
    case Red extends Color(0xFF0000)
    case ArbitraryColor(x: Int) extends Color(x)
}

enum SimpleColor{
    case Blue
    case Green, Yellow, Orange
}
```

This example shows all the possible ways of defining values:

1. A value can be parametrized using "extends" (like with *Red* color). It can only extend the enum in which it is (in this example, *Red* can only extend *Color*).
2. A simple value can be defined either alone (*Blue*) or with others separated by a coma (*Green, Yellow, Orange*).

3. A value can have a constructor (*ArbitraryColor*).

If the enum takes parameters (like *Color* in this example), all of its values must extend it and provide arguments.
A value can never have a body.
In addition to values, an Enum's body can contain the exact same things as classes.

We decided to add support for these enumerations in a library which is called Scalameta. It makes possible reading, manipulating and generating Scala source code.
Scalameta uses the same program representation as a compiler and works in a similar way to produce it. Such representation is called an AST (Abstract Syntax Tree). To produce this AST, Scalameta tokenizes and parses the code (see Background section for more details). Scalameta AST contains a lot of information (describing precisely all the program's structures and how they follow each other) that makes possible to recover source code from AST which is not necessarily possible with a compiler AST because of some simplification or optimization early made. This is useful for pretty-printing the code with high fidelity or do formatting. Scalameta offers for instance the possibility to construct programmatically the AST and then outputs the Scala source code corresponding.

Building on top of our changes to Scalameta, we added formatting support for Dotty enumerations in Scalafmt. Scalafmt uses tokens and AST provided by Scalameta to reformat the code following some guidelines.

Finally, we developed a Visual Studio Code extension using Scalafmt to format code. The extension provides code formatting using Scalafmt directly from the editor. It launches a language server that handles requests. The server's implementation is based on LSP.

LSP (Language Server Protocol) is a protocol created by Microsoft used between editor (the client) and a language server. It simplifies the integration of features like auto-complete or, in this case, formatting in editors.

2 Background

2.1 Tokenizing and parsing

We will see the two operations Scalameta does on a program to have an internal representation of it.

1. The tokenizing (also called lexing) is the first operation that is done on the code. It takes as input the text file (i.e. a sequence of character) and transforms it into a sequence of tokens. A token is the smallest piece that composes a program i.e. a keyword ('while'), an identifier ('foo'), an integer constant ('1234'), etc. The set of tokens for a specific language is defined by its grammar so it is fixed and known. This step transforms a very primitive input (characters) into objects that can be manipulated programmatically.

2. The next operation performed is the parsing. Basically the parser will transform the sequence of tokens returned by the lexer into a tree data structure called an AST (Abstract Syntax Tree). This AST is a representation of the program on which operations can be performed.

Let us take the following simple program to illustrate how the *lexer* operates :

```
object Foo {}
```

Once passed through the *lexer*, the sequence of characters will give this sequence of tokens :

```
Seq(KwObject, Identifier(Foo), Leftbrace, Rightbrace)
```

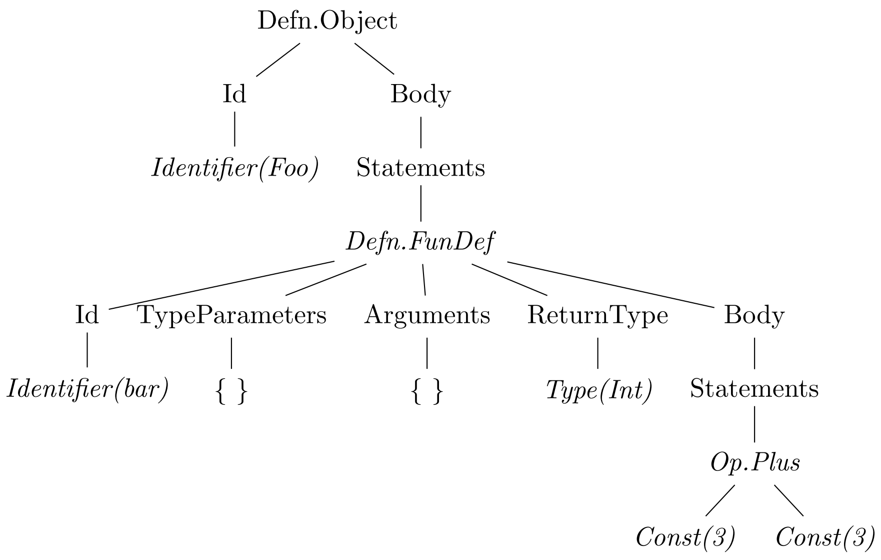
To see how AST looks like, let us take a more complex program :

```
object Foo {
    def bar(): Int = 3 + 3
}
```

The sequence of tokens for this program is :

```
Seq(KwObject, Identifier(Foo), Leftbrace, KwDef,
    Identifier(bar), Leftpar, Rightpar, Colon, Type(Int),
    KwEqual, Const(3), KwPlus, Const(3), Rightbrace)
```

The AST's structure is defined to be the most efficient for the work done by the compiler thus it is arbitrary and designed by the programmers in charge of the compiler. Here is a possible AST :



We can see that the program is represented in an abstract and structured way that let us operate on it more easily.
The structure of this AST is not the one used by Scalameta nor Dotty, it is a simplified one given only as illustration purpose.

3 Implementation

3.1 General idea

The stages of Scalameta's pipeline that had to be changed were: the lexer, the AST structure, the parser and the prettyprinter.
We will now develop in details the choices made for each modified stage and how they are implemented.

3.2 Lexer

The only token added is "enum". It is tokenized only when the dialect (i.e. the language level of Scala) is *Dotty* otherwise "enum" is tokenized as an identifier. The "case" token was already tokenized for the pattern-matching.

3.3 AST structure

Here are shown the classes introduced in the AST structure to support enumerations:

```
object Defn {
  //...
  @ast class Enum(mods: List[Mod],
                  name: scala.meta.Type.Name,
                  tparams: List[scala.meta.Type.Param],
                  ctor: Ctor.Primary,
                  templ: Template) extends Defn
                                   with Member.Type
  object Enum {
    @ast class Name(value: Predef.String @nonEmpty)
      extends scala.meta.Name with Term.Ref
    @ast class Case(mods: List[Mod],
                    name: Term.Name,
                    tparams: List[scala.meta.Type.Param],
                    ctor: Ctor.Primary,
                    inits: List[Init]) extends Defn
                                       with Member.Term {
      checkParent(ParentChecks.CaseEnum)
    }
    @ast class RepeatedCase(
                    mods: List[Mod],
                    cases: List[Enum.Name]) extends Defn {
      checkParent(ParentChecks.CaseEnum)
    }
  }
}
```

An Enum is very similar to a Class so its structure is the same. *Enum* is a class in the *Defn* object as it is a definition. The *Defn.Enum* contains a list of modifiers, a name, a list of type parameters, a constructor and template (the ”body” of the enumeration).

Concerning the Enum’s values, there are two types. Both of these types are represented by a class inside an *Enum* object.

1. *Defn.Enum.Case* that represents a value defined alone on its line. It can have a constructor, type parameters and an extends clause (like a Class without a body). For instance:

   ```
   case Foo
   case Bar[T](x: Int) extends Example(x)
   ```

   We decided that a value like *Foo* in this example is represented as a *Defn.Enum.Case* with no type parameters, no constructor and no extends clause (rather than a *Defn.Enum.RepeatedCases* with one single case).

2. *Defn.Enum.RepeatedCases* that represents values that are defined with one case keyword and separated by a coma. In this case values cannot have constructor, type parameter or extends clause. For instance :

   ```
   case Foo, Bar, Example
   ```

3.4 Parsing methods

Here is shown what changed in Scala’s grammar for enumerations:

```
EnumDef    ::= id ClassConstr ['extends' [ConstrApps]] EnumBody
EnumBody   ::= [nl] '{' [SelfType] EnumStat {semi EnumStat} '}'
EnumStat   ::= TemplateStat
             | {Annotation [nl]} {Modifier} EnumCase
EnumCase   ::= 'case' (id ClassConstr ['extends' ConstrApps]]
             |  ids)
```

We faced a difficulty when parsing enumerations’ values with the way Scalameta was dealing with whitespaces in the token list.
Originally, the list of tokens given to the parser was computed all in once at the beginning. This has as effect that the parser could not dynamically influence on the token list, especially on the whitespaces. Whitespaces are indeed removed in a lot of situations because Scala syntax is very permissive, i.e. sometimes a newline indicates the end of a statement (replacing a semicolon) or in other cases it is meaningless for the parser and is thus removed. This causes trouble when we tried to parse the values which begin with the keyword *case*. Indeed, before Dotty introduced enumerations, the keyword *case* was only used in pattern matching and in this situation every line breaks that are between the keyword *case* and the keyword ⇒ are removed for parsing.
For example the two following pieces of code produce the same list of tokens:

```
foo match {
    case Some(v)
        if(v == true) => true
}
```

```
foo match {
    case Some(v) if(v == true) => true
}
```

In the case of enumeration's values, the newline at the end is meaningful as it indicates the end of the statement.

We now compute the token list lazily. The parser fetches new tokens on demand and so can decide whether or not to discard some whitespaces depending on the context (parsing an enum or parsing a match case).

4 Applications

4.1 Scalafmt

We will now see a direct application of the Scalameta library. This application is Scalafmt that uses Scalameta to format Scala source code.

Scalafmt works at token level which means that it goes through the list of tokens and adds or removes some spaces and newlines between non-whitespace tokens to achieve good formatting. This is different from the prettyprinting that we saw earlier which only goes trough the AST and generates some code.

Scalafmt keeps the original code and just modified the spaces and newlines to reformat it. This method lets the formatter keep the comments and extra newlines the programmer can add. These parts of the program, that are not relevant for the program execution, are removed during lexing part and so are not represented in the AST. This has as consequences that the pretty-printer does not print them. For this reason, Scalafmt works with the tokens and uses information from the AST to know to which structure a token corresponds to (for example to know if a *case* token corresponds to a pattern matching's case or an enum's value).

We added a rule to format the *RepeatedCase*: if the length of the line exceeds the maximum column number defined for the document, it is formatted like this:

```
enum Foo {
  case
    Aaaaaaaaaaaaaaaaaaaaaaa,
    Bbbbbbbbbbbbbbbbbbbbbbb,
    Ccccccccccccccccccccccc
}
```

Otherwise it is formatted inline:

```
enum Foo{
  case A, B, C
}
```

The *Case* is formatted like a class's constructor:

```
enum Foo {
  case Bar(x: Int, y: Int, z: Int) extends Foo
  case Bar2(
      xxxxxxxxxxxxxxxxxxxxx: Int,
      yyyyyyyyyyyyyyyyyyyyy: Int,
      zzzzzzzzzzzzzzzzzzzzz: Int
  ) extends Foo
}
```

4.2 Visual Studio Code language server

To have a concrete application, we developed a language server for Visual Studio Code. A language server handles editor's requests and responds some information about the code for instance auto-completion or in this case new code's formatting.

The server implements the Language Server Protocol and can respond to only one request, the "Document Formatting Request". This request is sent from the client to the server to format a whole document. It contains the identifier of the document to be formatted and some formatting options. The server responds a list of modifications to apply to the document to format it. In this case, the server responds only one modification that replaces the whole code by the newly formatted one.

The language server is developed using *LSP4J* library and uses Scalafmt to format code. The server keeps trace of Scala documents that are open by storing the content of these files associated with the document's identifier given by the client. The server does not read the file when the client request a format, it uses the content it stores. When a file's content is changed, the client sends the new content to the server that can update its own representation of the file.

The server is then included in an extension written in TypeScript that can be installed on Visual Studio Code.

5 Conclusion

In this project, we updated Scalameta to make it handle a new feature Dotty introduced which is the enumerations. This implied to add new AST's structures to represent the new elements which are Enum and the values that belongs to it. To achieve that goal we modified the parser to compute the token list lazily.

After that, we updated another library that uses Scalameta to format Scala code: Scalafmt. Finally, we programmed and encapsulated in a extension a language server for Visual Studio Code that handles format request and uses Scalafmt to format the code and returns the output to the editor.