Metaprogramming

The following pages introduce the redesign of metaprogramming in Scala. They introduce the following fundamental facilities:

1. `inline` is a new modifier that guarantees that a definition will be inlined at the point of use. The primary motivation behind inline is to reduce the overhead behind function calls and access to values. The expansion will be performed by the Scala compiler during the `Typer` compiler phase. As opposed to inlining in some other ecosystems, inlining in Scala is not merely a request to the compiler but is a *command*. The reason is that inlining in Scala can drive other compile-time operations, like inline pattern matching (enabling type-level programming), macros (enabling compile-time, generative, metaprogramming) and runtime code generation (multi-stage programming).

2. Compile-time ops are helper definitions in the standard library that provide support for compile-time operations over values and types.

3. Macros are built on two well-known fundamental operations: quotation and splicing. Quotation converts program code to data, specifically, a (tree-like) representation of this code. It is expressed as `'{ ... }` for expressions and as `'[ ... ]` for types. Splicing, expressed as `${ ... }`, goes the other way: it converts a program's representation to program code. Together with `inline`, these two abstractions allow to construct program code programmatically.

4. Runtime Staging Where macros construct code at *compile-time*, staging lets programs construct new code at *runtime*. That way, code generation can depend not only on static data but also on data available at runtime. This splits the evaluation of the program in two or more phases or ... stages. Consequently, this method of generative programming is called "Multi-Stage Programming". Staging is built on the same foundations as macros. It uses quotes and splices, but leaves out `inline`.

5. Reflection Quotations are a "black-box" representation of code. They can be

parameterized and composed using splices, but their structure cannot be analyzed from the outside. TASTy reflection gives a way to analyze code structure by partly revealing the representation type of a piece of code in a standard API. The representation type is a form of typed abstract syntax tree, which gives rise to the `TASTy` moniker.

6. TASTy Inspection Typed abstract syntax trees are serialized in a custom compressed binary format stored in `.tasty` files. TASTy inspection allows to load these files and analyze their content's tree structure.

Table of Contents

- Inline
- Compile-time operations
- Macros
- Runtime Multi-Stage Programming
- Reflection
- TASTy Inspection
- The Meta-theory of Symmetric Metaprogramming

‹ Relatio... Inline ›