UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS700 Research

Design Pattern Metaprogramming Foundations in Rust

Abstract Factory and Visitor

Student number: u19239395

Supervisor:
Dr. Linda Marshall

??? 2020

Abstract

Keywords:

1 Introduction

2 Metaprogramming

Metaprogramming is a program that writes another program. A program operates on data; a metaprogram treats a program as its data [LS19, Ang17, She01]. Input to a metaprogram will be referred to as meta-code. This makes a metaprogram like any regular program. Therefore, a metaprogram can be refactored, abstracted, turned into library helpers, and tested [LS15]. Being able to write a program using code opens up many uses.

2.1 Uses

There are three main uses for metaprogramming: code optimization, code reuse, and analysis.

Code optimization A metaprogram can be used to improve the execution speed of the code. For example, with a Just-In-Time (JIT) compiler, a metaprogram can optimize blocks of code that are called more often than others [Hin13] or by caching the results of a method's call [MSD15]. Another example is the use of Domain-Specific Languages (DSL). Here a metaprogram has a better understanding of the code and can apply optimizations unknown to the compiler. Optimizations include knowing a value can never be negative [Hin13], simplifying an expression [She01], or offloading to the GPU [VPG$^+$18].

Code reuse Metaprogramming can also be viewed as a code reuse tool. Repeated code - like design patterns [LS15, Ale01] - can be wrapped behind a metaprogram function that will write the reusable code [LS19, KN19]. Complex code can also be translated from a simpler language that end-users can understand [Hin13]. The generated code can be anything from one-liners to classes [LS19].

Analysis Reading a program as input is the last use for metaprogramming. After reading a program, the metaprogram can analyze its control flow, check types on a dynamic language, or build a proof theorem [She01].

2.2 Dimensions

Metaprogramming has many dimensions. This section will briefly discuss these dimensions, as presented by Lilis and Savidis [LS19], by focusing on the relationship between the metalanguage and the object language, the model used for metaprogramming, when the metaprogram is executed, the location of the meta-code, and how the final program is represented.

2.2.1 Relationship to the object language

Metaprogramming will output code in some language. The output language is called the object language, while the metaprogram is written in the metalanguage. These two languages can be may be different. If they are different, the system is called heterogeneous. For a heterogeneous system, the metalanguage can extend the object language with extra features or be a completely new language. When they are the same, it is called a homogenous system [She01].

2.2.2 Model

Programming comes in different models, such as procedural, functional, and Object-oriented. The same is true for metaprogramming, which includes the following.

Macro systems A macro system takes input and expands it to some output. The output can be another macro. Thus expansion continues until no more macros are left.

The input can come in two forms. First is *lexical macros*, where the input is a stream of tokens. These tokens can be anything and do not need to adhere to a syntax. The second form operates on a specific syntax and is called *syntactic macros*.

Parsing the input can be procedural or pattern-based. Procedural will use an algorithm to generate the output. Patterns will match an input pattern to its output.

Reflection systems Reflection is the process an object follows to look at itself and then modify its structure accordingly. This is commonly done at run-time but can also happen at compile-time.

Metaobject Protocol (MOP) Rather than modifying an object's structure, MOPs modify an object's behavior. This is done by inheriting from a metaclass that can modify its own behavior. The modification then affects the subclasses [LZ95]. MOPs can also be used to modify a language's behavior [MSD15].

Aspect-Oriented Programming (AOP) Assume one wants to add logging or performance metrics to all functions in a program. Modifying each function will add a responsibility that does not form part of the function's duties. There is also the cost in the time it will take to modify each function. AOP will *weave* the extra responsibility - called *advice* - into each function.

Generative Programming Generative programming is like a macro system. The difference being that generative code is clearly not meta-code to be expanded by the macro system. They also typically represent their data as an Abstract Syntax Tree (AST).

Multistage Programming Multistage programming allows object code to be generated in stages. The generations can either be automatic or require manual annotations [She01, Tah04].

2.2.3 Metaprogram execution

Three options exist for when the metaprogram can be executed.

Before compilation The metaprogram can be executed before compilation. This offers the option of using any language for metaprogramming. The metaprogram takes a source file with meta-code and outputs a file without meta-code.

During compilation Running the metaprogram as part of the compilation is another option. This requires the compiler to support metaprogramming, or it needs to have a plugin for metaprogramming.

Run-time Lastly, the metaprogram might execute at run-time. This will require the language execution system to support dynamic code generation and execution.

2.2.4 Meta-code location

The next dimension is the location of the meta-code. Two options exist.

External The meta-code can be in an external file and will result in a new file with the object code. This option is used with the before compilation option and generative model. Alternatively, suppose this option is used with compilation time execution. In that case, the file needs to be passed to the compiler with a flag.

Embedded The meta-code can also be embedded within the program to transform. This means a source file with a mixture of regular code and meta-code. Embedded code can have three levels of context-awareness.

- Completely unaware: The meta-code only relies on the inputs passed to it. The code immediately after the meta-code is not available to the metaprogram.
- Local awareness: The meta-code relies not only on inputs but also on the code immediately after the meta-code.
- Global awareness: The meta-code relies on inputs and is aware of all code in the file.

2.2.5 Data representation

The final dimension to consider is the representation used to hold the final code. Since the final code is the metaprogram's data [B^+99], it needs to be held in some type. Many systems use strings, graphs, or an algebraic data structure [She01]:

String The final program is held in a string. This option is not desired since building a class may need hundreds of string append operations spanning hundreds of lines. The object code will be interleaved with the metaprogram making it hard to distinguish between them. This makes it easy to construct a string that is not syntactically correct.

Graphs A graph type will add structure to the program being built. Furthermore, it makes a better separation between the object code and the metaprogram code. However, it still does not guarantee that the structure will be syntactically correct.

Algebraic Storing the data as an algebraic expression with type encoding or an AST is the only guarantee of a syntactically correct program. However, building an AST by hand is hard. Lisp solved this problem by using *quasiquotes* [B$^+$99].

Quasiquotes is a form of templating/string interpolation. It allows writing the data as a "string" (enclosed in backtick quotes) in the object language's syntactical form. This *quoted* "string" is then transformed into the desired data structure. Thus, it acts as a shortcut for constructing an AST [LS15]. Placeholders are placed in the *quoted* "string" to be replaced with variables from the metaprogram context. These placeholders need to be identifiable. Thus, the placeholders are preceded by some *unquote* character [B$^+$99].

Having identified all the dimensions, let us identify the elements Rust uses for metaprogramming.

2.3 Metaprogramming in Rust

Rust has two metaprogramming functionalities build into the language. The first has been with the language for some time and is meant for general metaprogramming. The second is a newer addition added in late 2018 [1]. It is called *Procedural Macros* and is the only focus of this report [KN19].

The metalanguage for *Procedural Macros* is Rust. Thus, *Procedural Macros* are homogenous. From the name *Procedural Macros*, it is also clear it follows the macro model, and the parsing is procedural. The input stream is also a lexical token stream.

Execution happens during compilation. This means the macros need to be available to the compiler and thus need to be precompiled. Therefore, they need to be isolated from client code in a library marked for macro use [2]. The macro invocation is then embedded in the client code. Since *Procedural macros* come in three flavors, it has both local awareness or no awareness depending on the flavor used. The data representation is the same as the input - a lexical token stream.

The input token stream does not need to be Rust code, but only the output stream needs to be valid Rust code. Rust tries to keep its standard library as slim as possible while offloading features to libraries. Given the wide range of possible inputs, no standard library helpers exist for working with a token stream. However, two Rust libraries do exist for working with tokens streams.

The first is *syn*[3] for parsing Rust syntax to a syntax tree. Other parsers can also be build using *syn*.

The second is *quote*[4] for generating a token stream from Rust syntax. It is

[1]https://blog.rust-lang.org/2018/12/21/Procedural-Macros-in-Rust-2018.html
[2]https://doc.rust-lang.org/reference/procedural-macros.html
[3]https://docs.rs/syn/1.0.31/syn/index.html
[4]https://docs.rs/quote/1.0.7/quote/index.html

a macro that uses the quasiquotes concept from Lisp. Thus, anything in the *quoted* "string" is correctly highlighted, formatted, and autocompleted by an editor.

Let us discover the *TokenStream*, *syn*, and *quote* by exploring the three flavors of *Procedural Macros*.

2.3.1 Procedural Macro flavors

The three flavors of Procedural Macros are function-like, derive, and attribute macros.

Function-like macros Function-like macros are the most straightforward flavor of procedural macros. They take an input stream and return an output stream - i.e., they are context unaware. Listing 1 shows a reflective function-like macro that returns its input as is.

```
1  extern crate proc_macro;
2  use proc_macro::TokenStream;
3
4  #[proc_macro]
5  pub fn reflect(input: TokenStream) -> TokenStream {
6      input
7  }
8
9  reflect!(2 + 3, 5);
```

Listing 1: Reflective function-like macro

Lines 1 and 2 import the *proc_macro* library and the *TokenStream* type in the library. These two lines will be needed for all macros and will not appear in future examples. The *#[proc_macro]* attribute on line 4 marks the function that follows as a function-like macro. Line 5 shows it taking one input and returning one output - both of type *TokenStream*. On line 6, the input is returned unaltered. Line 9 shows the line that will appear in client code to call the macro using the same function name. All function macros are invoked using the *!* (exclamation) sign - called the *macro invocation operator* - to distinguish them from regular function calls. Line 9 will be replaced with the output "2 + 3, 5". Since the output is invalid Rust code, the compiler will give an error on line 9.

Notice how everything inside the parenthesis (2 + 3, 5) will be passed to input. A *TokenStream* can be thought of as a list of tokens. There are four possible token types [5]:

- An *Ident* to hold an identifier like a variable name or keyword.

[5] https://doc.rust-lang.org/proc_macro/enum.TokenTree.html

- A *Punct* to hold a single punctuation mark.
- A *Literal* to hold a literal like an integer value, string, or literal character.
- A *Group* to hold an inner/nested *TokenStream* surrounded by brackets.

Thus the 2, 3, and 5 will be literal tokens. The +(plus) sign and ,(comma) will be punctuations in both streams. Again, parsing and generating the list will be hard. Thus, the next example shows how to use the *syn* and *quote* libraries to make this easier.

Derive macros Derive macros are used to add methods to objects as seen in Listing 2.

```
1  use syn::{parse_macro_input, DerivedInput};
2
3  #[proc_macro_derive( GetType )]
4  pub fn derive_answer_fn( tokens: TokenStream ) -> TokenStream
   ↪ {
5      let context = parse_macro_input!(tokens as DerivedInput);
6      let name = context.ident;
7
8      let output = quote! {
9          impl GetType for #name {
10             fn get_type(&self) -> str {
11                 stringify!(#name)
12             }
13         }
14     }
15
16     output.into()
17 }
18
19 #[derive( GetType )]
20 struct SomeStruct;
21 impl GetType for SomeStruct { ... }
```

Listing 2: Derive macro example

Line 1 shows the import for the *syn* library to parse the input list of tokens to a syntax tree. Derive macros have the *proc_macro_derive* attribute followed by the macro name as seen on line 3. The input on line 4 is the context the macro is called on. Thus, derive macros have local context-awareness. The context is line 20.

Line 5 parses the context to a *DerivedInput* syntax tree from *syn*. *Syn* will give a compilation error if the parsing fails. Getting the struct name happens on line 6.

Lines 8 to 14 show the use of the *quote* library for quasiquotes. Rather than a *quoted* "string", it uses the *quote* macro. Placeholders are *unquoted* with the # (pound) sign. Thus, *#name* will come from the metaprogram context - line 6. Notice the syntax highlighting being correct inside the quote macro. Finally, line 16 converts output to a *TokenStream*.

The macro is used on line 19 with the derive attribute on an object. The compiler will write line 21. Thus derive macros append code below the annotated type.

Attribute macros Attribute macros are like function-like macros with context-awareness. Therefore, two token streams are passed to them as seen in Listing 3.

```
1  extern crate proc_macro;
2  use proc_macro::TokenStream;
3
4  #[proc_macro_attribute]
5  pub fn reflect_two(attr: TokenStream, item: TokenStream)
    ↪  -> TokenStream {
6      item
7  }
8
9  #[reflect_two(multiple => tokens)]
10 fn some_function() {}
```

Listing 3: Attribute macro example

This time the attribute above the function is *proc_macro_attribute*. The function name serves as the attribute name where it is used on line 9. The input is the first stream passed to the function, while the context is the second. Like function macros, the context on line 10 will be replaced with the output.

Since the metalanguage is Rust code, it is time to learn more about Rust.

3 Reporting

4 Conclusion

References

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[Ang17] Wisnu Anggoro. *Learning C++ Functional Programming*, chapter 6, pages 171–199. Packt Publishing Ltd, 2017.

[B+99] Alan Bawden et al. Quasiquotation in lisp. In *PEPM*, pages 4–12. Citeseer, 1999.

[Hin13] Konrad Hinsen. A glimpse of the future of scientific programming. *Computing in Science & Engineering*, 15(1):84–88, 2013.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018).* No Starch Press, 2019.

[LS15] Yannis Lilis and Anthony Savidis. An integrated implementation framework for compile-time metaprogramming. *Software: Practice and Experience*, 45(6):727–763, 2015.

[LS19] Yannis Lilis and Anthony Savidis. A survey of metaprogramming languages. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.

[LZ95] Arthur H Lee and Joseph L Zachary. Reflections on metaprogramming. *IEEE Transactions on Software Engineering*, 21(11):883–893, 1995.

[MSD15] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 545–554, 2015.

[She01] Tim Sheard. Accomplishments and research challenges in metaprogramming. In *International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer, 2001.

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[VPG+18] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. Boast: A metaprogramming framework to produce portable and efficient computing kernels for hpc applications. *The International Journal of High Performance Computing Applications*, 32(1):28–44, 2018.