Programming Languages Project

Exploring Meta-Programming

Danesh Toshniwal
Neil Handa
Kadaru Jashwanth Reddy

April 16, 2024

Abstract

Meta-programming refers to the practice of writing programs that manipulate other programs (or themselves) as their data. It is a powerful technique used in various programming paradigms such as functional programming, object-oriented programming, and logic programming. In this project, we aim to delve into the concept of meta-programming, exploring its different aspects, techniques, and applications.

The project seeks to:

- Analyze the meta-programming features of languages like Ruby and Lisp.
- Investigate the implications of meta-programming on programming language design, focusing on syntax, semantics, and language features that facilitate or hinder meta-programming techniques.
- Explore the role of reflective capabilities or meta-object protocols in enabling meta-programming.
- Examine how meta-programming interacts with the type systems of programming languages, including challenges related to type safety and type inference in dynamically generated code.

Contents

Chapter 1

Preface

1.1 Solution Outline

The anticipated deliverables of this project include:

- A comprehensive study report detailing the findings and analysis of meta-programming features in languages such as Ruby and Lisp, as well as their impact on language design.
- Code examples demonstrating various meta-programming techniques in selected languages.
- An algorithmic exploration of reflective capabilities and meta-object protocols, showcasing their role in enabling meta-programming.
- Conceptual frameworks discussing the interaction between meta-programming and type systems, including challenges and potential solutions.

1.2 Main References

1. **Blog on Ruby Metaprogramming:** optymize.io/blog/ruby-metaprogramming
2. **Conference paper: Metaprogramming in Ruby: a pattern catalog:** Link to paper (still awaiting access)
3. **Taxonomy of the Fundamental Concepts of Metaprogramming** A taxonomy paper on metaprogramming concepts
4. **Metaprogramming Ruby 2: Program Like the Ruby Pros** a online resource on metaprogramming concepts with Ruby 2.0

1.3 Team's Progress

Our team has made significant progress in understanding the fundamentals of meta-programming and exploring various techniques related to it, in various programming languages. We have completed a literature review and identified key areas for further investigation. Challenges encountered include the complexity of certain meta-programming concepts and the selection of appropriate case studies.

1.4 Work Distribution

The remaining work within our team is distributed as follows:

- Danesh Toshniwal: Responsible for researching meta-programming techniques in functional programming languages (Ruby, Lisp, etc.)
- Neil Handa: Tasked with investigating meta-programming and its implications on Programming Language Design.
- Kadaru Jashwanth Reddy: Assigned explore the role of reflective capabilities in enabling meta-programming..

Chapter 2

Introduction to Meta Programming & its features in languages

2.1 What is meta programming? What are the advantages?

Meta-programming is a programming technique that empowers developers to write code that can manipulate or generate other code dynamically during runtime. This advanced approach allows for the creation of more flexible, adaptable, and efficient software systems by leveraging the introspective capabilities of programming languages. By treating code as data and manipulating it programmatically, meta-programming enables tasks such as automating repetitive tasks, implementing domain-specific languages, and enhancing code reusability and maintainability.
We will see few of these in the upcoming topic.

2.2 Meta Programming features in Ruby

- **General Idea of Metaprogramming in Ruby:** "Code writing", because the code written in Ruby is dynamically enhanced with some additional code at runtime by the programmed code itself.
- **Concept of Introspection:** As mentioned above, the codes in Ruby metaprogramming are modified at the runtime. So, the code can be interacted with, or the user can ask question to the code or manipulate it.

main.rb

```ruby
class MyClass
  def greet
    puts "Hello!"
  end
end

# Creating an instance of MyClass
obj = MyClass.new

# Check if the instance responds to the greet method
if obj.respond_to?(:greet)
  puts "MyClass can greet!"
else
  puts "MyClass cannot greet!"
end

# Check if the instance responds to the farewell method
if obj.respond_to?(:farewell)
  puts "MyClass can bid farewell!"
else
  puts "MyClass cannot bid farewell!"
end
```

```
MyClass can greet!
MyClass cannot bid farewell!
```

Figure 2.1: Introspection in Ruby

- **Open Classes or Monkey Patching:** Monkey patching in Ruby refers to the practice of dynamically modifying or extending existing classes during runtime, allowing developers to add, modify, or remove methods from core classes to suit specific application requirements.

```ruby
class Integer
  def even?
    self % 2 == 0
  end
end

# The predefined class "Integer" is being extended with an additional method
puts 4.even?  # true
puts 5.even?  # false
```

input

```
true
false
```

Figure 2.2: Monkey Patching in Ruby

- **Using Send in Ruby:** The 'send' method in Ruby allows for dynamic method invocation by accepting the method name as an argument, facilitating flexible method calls and interaction with objects.

- **Class Definitions:** In Ruby, class definitions are executable code, enabling dynamic behavior during class creation, such as executing statements or defining methods within the class body.

- **Dynamically Defined Classes and Methods:** Ruby enables the creation of classes and methods dynamically during runtime using constructs like 'Class.new' and 'define_method', providing flexibility and adaptability in code structure and execution.

2.3 Meta Programming features in Lisp

- **General Idea:** Lisp's metaprogramming uses homoiconicity, allowing code and data to share the same syntax, enabling manipulation of program structure.
- **Introspection:** Lisp enables examining and modifying program structure at runtime, aiding in functions, symbol, and macro inspection.
- **Macros:** Define new language constructs or modify existing ones, reducing boilerplate and enabling domain-specific languages.
- **Code as Data:** Lisp represents code as lists, enabling manipulation similar to other data structures.
- **Read-Time Evaluation:** Lisp executes code during reading, allowing for computations and manipulation during that phase.
- **Reflection:** Lisp provides functions for runtime introspection on functions, symbols, and macros.

What is planned for the second half of the project?

More detailed analysis of each of the meta programming features, as well as finding more instances of meta programming in other languages.

Chapter 3

Implications of meta-programming on PL design

3.1 Introduction

Now that we know Meta-programming transcends the traditional boundaries of programming by allowing programs to introspect and modify themselves, this capability opens doors to a wide array of advanced techniques such as code generation, domain-specific language creation, and aspect-oriented programming.

However it is important to note that the efficacy of meta-programming heavily depends on the design choices made in the underlying programming language.

Syntax and semantics play crucial roles in determining the expressiveness and ease of use of meta-programming facilities, while language features serve as the building blocks that enable or constrain meta-programming capabilities. We will now look the implications of meta-programming on programming language design, shedding light on how to harness the full potential of meta-programming techniques.

3.2 Syntax

- **Expressiveness:** A language with a concise and flexible syntax can make it easier to write expressive meta-programming code. For example, languages like Ruby and Lisp have expressive syntaxes that enable powerful meta-programming techniques such as code generation and DSLs.

- **Macro Systems:** Languages that provide built-in support for macros, such as Lisp's macros or Rust's macro system, offer developers a powerful tool for meta-programming. Macros allow developers to define custom syntax and transformations, enabling advanced meta-programming techniques.
- **Readability:** Complex meta-programming code can sometimes be difficult to read and understand. A language with clear and intuitive syntax can mitigate this issue, making meta-programming code easier to comprehend and maintain.

3.3 Semantics

- **Reflection:** Languages with robust reflection capabilities provide developers with powerful tools for meta-programming. Reflection allows programs to introspect and modify their own structure and behavior at runtime, enabling dynamic code generation, adaptation, and customization.
- **Metaprogramming Abstractions:** Some languages offer high-level abstractions for meta-programming, such as metaclasses in Smalltalk or metaprogramming libraries in Ruby. These abstractions simplify common meta-programming tasks and promote code reuse and maintainability.
- **Safety:** Meta-programming can introduce safety concerns, such as the risk of runtime errors or unintended behavior. Languages with strong type systems or static analysis tools can help mitigate these risks by providing compile-time checks and ensuring type safety in dynamically generated code.

3.4 Features

- **First-class Functions and Closures:** Languages that treat functions as first-class objects and support closures can facilitate advanced meta-programming techniques such as higher-order functions, function composition, and aspect-oriented programming.

- **Dynamic Typing:** Dynamic languages like Python and Ruby, which employ dynamic typing, provide flexibility for meta-programming but can also introduce challenges related to type safety and debugging.
- **Compile-time Metaprogramming:** Meta-programming can introduce safety concerns, such as the risk of runtime errors or unintended behavior. Languages with strong type systems or static analysis tools can help mitigate these risks by providing compile-time checks and ensuring type safety in dynamically generated code.

In summary, the design of programming languages significantly impacts the ease and effectiveness of meta-programming techniques. By considering factors such as syntax expressiveness, semantic features, language abstractions, and tooling support, language designers can create environments that empower developers to harness the full potential of meta-programming for building expressive, adaptable, and maintainable software systems.

Chapter 4

Meta-Programming and Reflective Capabilities of Meta Programs

"Reflective capabilities or meta-object protocols allow programs to inspect and modify their own structure and behaviour."

4.1 Introduction

Meta-programming allows programs to inspect and modify their own structure, and dynamically generate code all at runtime. These capabilities are pivotal features of languages like Ruby 2.0. Programmers who use Ruby, tend to use meta-programming features of the language more often than not, to tackle code duplication, dynamic code generation, class wrappers, function wrappers, etc.

In this chapter, we will look into some of these meta-programming features in detail and discuss how a program's ability to inspect itself and modify itself, which is commonly referred to as introspection & reflection in meta-programming jargon, comes into play in these features.

4.2 Introspection

Think of your source code as a world teeming with vibrant citizens: variables, classes, methods, and so on. These entities (citizens) are technically known as *language constructs*.

In many programming languages, language constructs in a source code completely disappear during runtime. For example, languages like C, C++,

once the compiler has finished its job of converting source code to object code (machine code), things like variables, classes, and methods have lost their concreteness. They are no longer present or defined in the machine code explicitly, but their behaviour is captured and is present in the form of data stored in memory locations of computer. The language constructs in these programming languages are like Ghost citizens, during runtime. They are there, but not really.

As opposed to this, in languages like *Ruby, Python, and Lisp*, runtime is much more lively, with language constructs still preserved. The main difference between these languages that allows this to happen, is the fact that, the later stated class of programming languages are all interpreted, and not compiled programming languages. Compilation is like a brick wall that disallows language constructs to pass through into the runtime execution of the program. Because of this subtle difference, we can ask a ruby program questions directly about itself during runtime, such as the class name of an object, method signature, scope, etc. This is called ***introspection***.

Introspection is the feature of a programming language, which allows us to query (or) refer to the language constructs, used in the program itself, during runtime.

```ruby
class Greeting
    def initialize(text)
        @text = text
    end

    def welcome
        @text
    end
end

my_object = Greeting.new("Hello")

my_object.class # => Greeting

my_object.class.instance_methods(false) # => [:welcome]

my_object.instance_variables # => [:@text]
```

4.3 Reflection

Reflection refers to the property of a meta-program to write (or) generate code at runtime, that is non-existing before runtime (or) compile-time. This is a powerful feature that allows programmers to write code that automatically writes boiler-plate code, and helps programmers avoid what is called code duplication. Code duplication is dreaded in programming due to its nature of duplicating bugs along with the code, and making code unreadable (or) boringly long. Reflective capabilities of meta-programs provide a creative way to tackle this problem and keep the code DRY.

Here is an advanced example from Ruby programming that shows Reflection in action. Let us consider the problem of mapping objects to classes, which is a frequently occurring task in web-development projects. Consider the following code that maps *Movie* class to *Movies* table, which has the attributes *title* and *director*.

```
18 class Entity
19     attr_reader :table, :ident
20     def initialize(table, ident)
21         @table = table
22         @ident = ident
23         Database.sql "INSERT INTO #{@table} (id) VALUES (#{@ident})"
24     end
25
26     def set(col, val)
27         Database.sql "UPDATE #{@table} SET #{col}='#{val}' WHERE id
               =#{@ident}"
28     end
29
30     def get(col)
31         Database.sql ("SELECT #{col} FROM #{@table} WHERE id=#{
               @ident}")[0][0]
32     end
33 end
34
35 class Movie < Entity
36     def initialize(ident)
37         super "movies", ident
38     end
39
40     def title
41         get "title"
42     end
```

```
43
44      def title=(value)
45          set "title", value
46      end
47
48      def director
49          get "director"
50      end
51
52      def director=(value)
53          set "director", value
54      end
55  end
```

Entity is super-class and tries to define the generic SQL query methods and attributes. While Movie inherits Entity, i.e., a subclass of Entity. Movie defines getter and setter for each of the attributes with the help of set and get methods of superclass. Now, we can see that same code is being repeated multiple times, with the only difference being attribute name.

Now, we see how a popular library in ruby called Active Record library tackles this problem by using reflective capabilities of Ruby language under the hood.

```
56  class Movie < ActiveRecord::Base
57      # Do nothing here. It's as simple as that !
58  end
59
60  movie = Movie.create
61  movie.title = "Doctor Strangelove"
62  movie.title # => "Doctor Strangelove"
```

Now, the attributes title and director, and the getters and setters for these attributes are being automatically inferred and generated at runtime, by looking at the corresponding Movies table schema in the database. There is code inside Active Record library, that adds attributes of the table and their corresponding methods at runtime when the class is run.

4.4 Dynamic Methods, Open Classes, Monkey Patching, prepend

All the above-named features (or) concepts involve introspection and reflection. Let us discuss them one by one with examples:

4.4.1 Open Classes

Firstly, classes in ruby are very different from classes in other programming languages. In ruby classes are ran at runtime and not just stored in memory.

Consider the following example:

```
3.times do  # executes the block inside, 3 times.
    class C
        puts "Hello"
    end
end
```

```
<   Hello
    Hello
    Hello
```

In the above example code, the class C is defined only once and it has only one definition, however the the definition is repeated 3 times. Also, note that, inside class C, we do not have any meaningful things, but a statement to be executed ‘puts‘. In python, a programming like this would throw error. However, in Ruby, classes are executed at runtime and can contain executable code.

This leads to the concept of Open Classes. Unlike in traditional programming languages like Java, python, Ruby allows defining the same class with same name multiple times. All subsequent definitions of same class, after the initial definition are considered as an addition to existing class definition. See the example below to understand this:

```
class D
    def x; 'x'; end
end

class D
    def y; 'y'; end
end

obj = D.new
obj.x # => "x"
obj.y # => "y"
```

In the above code, the initial definition for class D only declares 'x' variable. The later definition declares 'y' variable, which is added to class D.

This is a subtle instance of the reflection, as the class code is ran during runtime, and there is no compilation step in between.

4.4.2 Monkey Patching

The addition of bits and pieces of code to a class definition, as shown in the previous section, is called ***monkey patching***. Patching refers to the act of adding a new method or code to an existing class definition to solve some new problem, each time. The prefix "Monkey" was given because this kind of patching could potentially lead to errors. As a class is open to extension not just by itself but also by other classes during runtime means that there could be collisions in method names, as new methods are added to the class definition.

Given below is an example:

```
class Array
    def replace(original, replacement)
        self.map {|e| e == original ? replacement : e }
    end
end
```

test becomes as follows:

```
def test_replace
    original = ['one', 'two', 'one', 'three']
    replaced = original.replace('one', 'zero')
    assert_equal ['zero', 'two', 'zero', 'three'], replaced
end
```

and the output is:

```
92 [].methods.grep /^re/ # => [:reverse_each, :reverse, ..., :replace,
       ...]
```

This happens because the core class Array (array class in Ruby, for array data structure) already has a method named replace defined. Since, Array class allows extension, it allowed us to define another method with the same name, causing a name collision which lead to an error.

This can be avoided by checking if the method already exists on the class, as follows:

```
93 Array.methods.grep(/re/) # => [:reverse_each, :reverse, ..., :
       replace, ...]
```

This again falls under introspection. We are asking the class or object to get a list of methods defined on it, at runtime.

4.4.3 send:

There are different ways of calling a method in Ruby, static and dynamic. The dynamic way of calling allows us to decide which method to call, at runtime and also lets us define when to call it programmatically.

See the code below:

```
94 class MyClass
95     def my_method(my_arg)
96         my_arg * 2
97     end
98 end
99
100 obj = MyClass.new
101 obj.my_method(3) # => 6
102
103 obj.send(:my_method, 3) # => 6
```

The *send*() method takes the method name and arguments as parameters and sends a signal to the object to call the method name, if it exists.

4.4.4 Dynamic Methods

Ruby also allows you to define methods dynamically using Classdefine_method. You just need to provide a method name and a block, which becomes the method body:

```
104 class MyClass
105     define_method :my_method do |my_arg|
106         my_arg * 3
107     end
108 end
109
110 obj = MyClass.new
111 obj.my_method(2) # => 6
112
113 require_relative '../test/assertions'
114 assert_equals 6, obj.my_method(2)
```

define$_m ethodisexecutedwithinMyClass, somy_m ethodisdefinedasaninstancemethodo$

4.4.5 Prepend

Prepend includes a Module or class below the current class in the ancestor tree. This allows us to change the ancestor tree of classes or individual objects at runtime, by passing arguments to the constructor.

This is an example usage of prepend, with the resulting class diagram:

```
115 Module M2
116     ...
117 end
118
119 class C2
120     prepend M2
121 end
122
123 class D2 < C2; end
124
125 D2.ancestors # => [D2, M2, C2, Object, Kernel, BasicObject] -
    Ancestor Tree of D2
```

These are a few practical Meta-programming features of Ruby and their usage, and syntax. We wish to cover more features and dive deeper into how introspection and reflection can be used to code interesting problems and make a programmer's life easy by avoiding code repetition.

End of Document