

---

# AOYAMA GAKUIN UNIVERSITY INTERNSHIP PROGRAM: DOMAIN IN PROGRAMMING

---

Akira Uchiyamada — R#11351000  
Aoyama Gakuin University  
Computer Science  
August 7, 2018

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Details . . . . .	2
1.2	What I've learned . . . . .	2
<b>2</b>	<b>Research</b>	<b>3</b>
2.1	Ruby Kaigi . . . . .	3
2.2	Metaprogramming Ruby . . . . .	3
<b>3</b>	<b>Requirements</b>	<b>3</b>
3.1	Domain . . . . .	3
3.2	Features . . . . .	5
<b>4</b>	<b>Development</b>	<b>6</b>
4.1	Introduction . . . . .	6
4.2	Declaration of domains . . . . .	6
4.3	Assignment of signatures to function/methods . . . . .	9
4.4	Declaration of variables . . . . .	18
4.5	Combination of domains . . . . .	21
4.6	Implicit conversion of variables . . . . .	22
4.7	Type Checking at Compile Time . . . . .	25

# 1 Introduction

This summer, from May 31st to July 28th, I was in Japan to participate in an internship program hosted by Aoyama Gakuin University. There, I learned various different skills that I hope to use in the future as a computer scientist.

## 1.1 Details

This internship program, hosted by Aoyama Gakuin University in Tokyo, Japan, is a research based program that is aimed towards helping the students improve their skills. The program I participated in stated that I must learn and contribute to the programming language Ruby or Ruby on Rails in 2 months. The professor of the lab I was assigned to is Professor Martin Dürst, who is a computer scientist that specializes in optimizing Unicode in programming languages. He is also a committer of Ruby and frequently participates in conferences based on his skills, like Ruby Kaigi.

Since the job description was vague, I was told to come up with some project idea and implement them during the internship program. There, I worked on a project called domain, which is a type checking system based on the mathematic concept of domain. I chose this project because I am interested in the development of programming languages, and because I was provided with a chance to contribute to the programming language itself, I decided that it was a good time to challenge myself and implement something I always wanted to see in programming languages.

The timeline of this internship looked like the following:

- Trip to Sendai to attend Ruby Kaigi
- Researching about Ruby through books
- Coming up with a project to work on
- Writing an informal requirement document on features I wanted to add
- Implementing and testing the project as I go
- Final presentation towards every interns and professors that was involved in this program to demonstrate what I did

## 1.2 What I've learned

I've learned many things over this internship. First, I learned about Ruby. I learned about how the language works in the lower level, as well as very niche tools Ruby have to assist the programmers. I've also learned about how the research labs are operated. I attended weekly meetings to explain what I did that week and watched other students perform their own research. Finally, I've learned about Japanese culture and how they work as a team.

I think this experience will be beneficial to me in the future, as my ultimate goal as a computer scientist is to develop a programming language, which is something I partially did in this internship program. Additionally, because it was such an extensive project, I had to use various techniques I've learned as a student here, such as software development techniques, breadth first search algorithm, and finite state machine diagram, which helped me refresh my memory on how these concepts work. By applying concepts I learned from older courses, I think I've become a better student and better programmer overall.

## 2 Research

Before I began working on my project, I spent the beginning few weeks learning about the programming language Ruby. The learning process included going on a 3 day trip to Sendai to attend Ruby Kaigi and reading various books about the programming language Ruby.

### 2.1 Ruby Kaigi

The first thing I did when I started the intern was attending Ruby Kaigi. Ruby Kaigi is an annual conference for the programming language Ruby that hosts various talks by different programmers ranging from talking about their on going project to educational lectures about niche features of Ruby. In this conference, I have learned a lot about the language itself, such as its weaknesses and strengths and how people solve problems they encountered. The most important thing I've learned about Ruby in this conference is most likely Binding and TracePoint. Binding is an object representing the scope of the program. It can retrieve information on things such as value of local variable and even running some codes within that scope using `eval()` method. TracePoint is a set of events that trigger based on the code you are running. For example, TracePoint can trigger every line of your code, when a method has been called, when the method returns, and many other methods. It is intended to be used for debugging and logging, but like you can see in the future sections, it is a very powerful tool that can manipulate the code in the bizarre manners. Without some of the knowledge I gained through Ruby Kaigi I may not have been able to implement many of the features in this program.

### 2.2 Metaprogramming Ruby

After the Ruby Kaigi has ended, the internship has officially begun. The first thing I did when I got to the lab was to learn a bit more about the Ruby. Thanks to the supervising professor's recommendation, I was able to pick up on the principles of metaprogramming quickly. One book the professor recommended me was *Metaprogramming Ruby* by Paolo Perrotta. The book, like the title suggests, is about metaprogramming in Ruby. It explained several different features about metaprogramming in Ruby in detail, such as aliasing, hook methods, eugenclasses, and how the classes and inheritance are structured. This helped me learn more about the metaprogramming processes and other features that makes Ruby a powerful tool for DSL or domain specific languages. Through this book, I picked up metaprogramming techniques and DSL principles, which I put to use in my project. Although I have read other books like *Ruby Under Microscope* by Pat Shaughnessy, Metaprogramming Ruby was the most influential to me in this project.

Now that I have spent some time learning about how Ruby operates, it was time to think about exactly what kind of project I wanted to work on.

## 3 Requirements

### 3.1 Domain

After I spent several days learning about Ruby, I have began working on projects I was required to do for the internship.

Because I was interested in developing programming language in the future, I eventually decided to work on a type checking concept I have been thinking about for a long time. This type checking system is based

on the mathematical concept of domain. In mathematics, domain is a set of values the function can properly map. In other words, if the value is part of a domain's set, then the function can use that value.

$$\begin{aligned} \text{let } A &= \{x \mid x \in \mathbb{Z}, x \neq 0\} \\ f &: A \rightarrow \mathbb{Z} \end{aligned}$$

Since the function in computer science is similar to the functions in mathematics (hence the same name), it is not far-fetched to think that domains in mathematics can also be applied to the functions in computer science. With this in mind, take a look at functions in statically typed languages:

```
1 int func(int x, int y) {  
2     return x + y;  
3 }
```

Listing 1: Domain example

Notice how the arguments are restricted. Both arguments `x` and `y` require the `int` type. This can be interpreted as a form of domain, as it restricts the input to certain set of values. It is also the set of values the function is expected to work, which matches the definition of the domain nicely. Although the functions in statically typed languages contain similar quality as domains in mathematics, it is not very flexible. The arguments can only restrict based on the types of the value, even though there can be multiple ways to restrict them. For example, what if you needed a value that is specifically a `String` that contains an integer? What if the function should only accept values from 0 to 500? It is possible to impose such rules using `if` statements within function:

```
1 int divide(int x, int y) {  
2     if (y == 0) {  
3         // handle bad argument  
4     }  
5  
6     return x / y;  
7 }
```

Listing 2: Domain example

However, by doing such thing, the arguments lose its similarity with domain. The arguments explicitly states that arguments can be any `int`, yet it reacts badly to 0, which is a perfectly valid `int` value. Additionally, it does not respond well when the function can theoretically accept more than one types. Because of the reasons stated above, I felt that there should be a brand new types to accomodate these flaws. The pseudocode for the domain looks like the following:

```
1 # pseudocode in Python-like syntax  
2 domain non_zero_int:  
3     y = lambda a: a != 0  
4     rule(integer, y)
```

Listing 3: Domain pseudocode

In this implementation of domain, a domain is declared like a class. The domain then creates a scope inside, which is intended to allow programmers to write rules for that domain. This domain will allow programmers to flexibly define the domains for the functions. The domains can then be used wherever types are appropriate.

```
1 # pseudocode in Python-like syntax  
2 def f(int x, non_zero_int y):  
3     return x / y
```

Listing 4: Domain pseudocode

## 3.2 Features

Once I had a solid understanding of what exactly a domain is, I began jotting down ideas for what to implement. After few days of brainstorming, I have decided on six major features domains should have, which are declaration of domain, assignment of signatures, declaration of variables, combination of domains, implicit conversions, and type checking at compile time.

Declaration of domains is straight forward. In order to use the domain, one must be able to declare and define it. The ideal syntax should be something similar to the pseudocode given above. The domain should create its own scope of rules, and the syntax for rules should be intuitive and only take the domain to enforce the rules on and a function that determines if the object is valid.

Assignment of signatures is essentially what statically typed languages do. It restricts the argument and the return value the function can receive. This allows programmers to have more type security in the codes like the statically typed languages, while also giving the programmers more freedom on the type of arguments the functions should have.

Declaration of variable restricts the variables like the variables in statically typed languages:

```
1 # pseudocode in Python-like syntax
2 non_zero_int x
3
4 x = 500          # no error
5 x = 3000         # no error
6 x = 0           # error
```

Listing 5: Domain pseudocode

Having this feature should make it easier for the program to type check, because it would understand the domain of the object before runtime and detect some errors this way.

Combination of domain is exactly as it says. Since domain is a set of values in mathematics, domain in computer science should also be able to combine through set operations such as union, intersection, and difference

$$\begin{aligned} let A &= \mathbb{R} \cup \mathbb{Z} \\ B &= \mathbb{R} \cap \mathbb{Z} \\ C &= \mathbb{R} \setminus \mathbb{Z} \end{aligned} \tag{1}$$

In this case, A should accept a value that is either a real number or integer, B should accept a value that is both a real number and an integer, C should accept a real number that is not an integer. With this kind of feature, it will make some domains very trivial to define. Credit card number, for example, is a great example to use the set operation on. Credit card numbers should be stored in a string format because although credit card numbers are integers, they do not require any arithmetic operations performed on them, and 16 digit numbers has a chance of overflowing on int with low bit counts. However, you should also not be able to store every string in as credit card number. Credit card should always be a number, and any other string should be rejected. In this case, you can easily define such domain by making an intersection between integer and string like so:  $CreditCardNumber = Integer \cap String$ .

Implicit conversion is another feature domains should have. Because domains can hold more than one types, there must be a way to convert between all types domains can represent. For example, if the domain CreditCardNumber is an intersection between Integer and String, then it should theoretically respond to both Integer command (such as absolute value) and String command (such as length). For this to happen, there should be a way to translate between Integer and String.

```
1 domain CreditCardNumber:
2     A = Integer & String
3     rule(A, lambda x: true)
```

```

4 translation(Integer, String, lambda x: str(x))
5 translation(String, Integer, lambda x: int(x))
6

```

Listing 6: Domain pseudocode

With addition of translation rules, the value stored in CreditCardNumber domain can translate between Integer and String at will, and the domain will be able to automatically translate and use the correct type when necessary.

Finally, type checking at compile time should be the last feature domains should have. Because many of the type errors caught in the statically typed languages are on compile time, in order to compete with such powerful feature, domain should also be able to type check to some extent at compile time.

After all the features have been fleshed out, I began developing the project using Ruby.

## 4 Development

### 4.1 Introduction

In this section, I will discuss the development process of the domain project. There will be six sections, each exploring the six main features established in the requirement section, which are declaration of domains, assignment of signatures to function/methods, declaration of variables, combination of domains, implicit conversions of variables, and type checking at compile time.

### 4.2 Declaration of domains

The first feature I have implemented was declaration of domains. There are several methods and classes that are relevant to the code, which will be discussed one by one.

```

1 def create_domain(compound: 0, name: "")
2   if !block_given? && compound == 0
3     raise ArgumentError.new "No block of rules were given. There must be at
4     least one rule for the domain"
5   end
6
7   cl = Class.new do
8     extend DomainClass
9     include DomainClass
10  end
11
12  prev = @domain_created
13
14  @domain_created = cl
15
16  @domain_created.compound_domain = unless compound == 0 then compound else
17  @domain_created end
18  @domain_created.rules = {}
19  @domain_created.translators = {}
20  @domain_created.default = nil
21  @domain_created.translation_map = {}
22
23  yield if block_given?
24
25  @domain_created.translators.each_key do |key|

```

```

24     d_in, d_out = key
25
26     if !@domain_created.translation_map.has_key? d_in
27         @domain_created.translation_map[d_in] = []
28     end
29
30     @domain_created.translation_map[d_in] << d_out
31 end
32
33 @domain_created = prev
34
35 cl.send :generate_translators
36
37 if not name.empty?
38     if self.inspect == 'main'
39         Object.const_set(name, cl)
40     else
41         self.const_set(name, cl)
42     end
43 else
44     return cl
45 end
46 end

```

Listing 7: Method for creating domains

The first of the relevant methods is `create_domain`. This method is simple compared to some of the TracePoint driven methods that will appear in the later codes. First, the code checks to see if there are any rules with the provided domain. This is done to prevent users from create a domain that lacks any rules. Next, the method creates an anonymous class that implements `DomainClass`, which is a module that contains various basic functions for domains to function, which will be shown in Figure 2. Next, an instance variable `@domain_created` is stored in `prev` variable. This is because the `@domain_created` variable is basically a global variable for the `create_domain` method, and if another domain was declared within the block, the value can be overridden with the new domain. To prevent this, `prev` store the value `@domain_created` was previously and return it to the original value after the domain is created. After `prev` has been set, the anonymous class is assigned to `@domain_created` and all important values such as rules and translation rules are initiated. The block is then yielded to read all the rules inside, and saved to the hashes. The lines 23 to 35 will be explained in detail in the later sections, as it is more relevant there. Finally, the anonymous class is given a name and is defined in the appropriate classes as a constant.

```

1 module DomainClass
2     include DomainErrors
3     attr_accessor :rules
4     attr_accessor :translators
5     attr_accessor :compound_domain
6     attr_accessor :default
7     attr_accessor :translation_map
8
9     def print_rules; end
10
11     def print_translators; end
12
13     def value?(value); end
14
15     def check_rules(rules, value); end
16
17     def translate(d_in, d_out, value); end
18
19     def value=(value); end
20
21     def value(domain=nil); end

```



## Listing 8: Domain module

DomainClass module is the next set of codes that will be discussed. This module contains various different methods a domain should have to function properly. The implementations for each of the methods are omitted to prevent codes from taking too much space and because they are mostly self-explanatory. The most important methods are `value?`, `value=`, and `value`. `value?` method checks the value in the argument to make sure the argument follows the rules of the domain. `check_rules` in the module is the private helper method for the `value?` method. `value=` assigns the value to the domain for domain to use in the future, which can be read through the `value` method. Domain should also have ways to translate from one domain to another, which is used in implicit conversions.

```

1 class Object
2   class << self
3     include Util
4     def rules
5       [{ self => Proc.new { |x| true } }]
6     end
7
8     def compound_domain
9       self
10    end
11
12    def translators
13      {}
14    end
15
16    def value?(x)
17      return x.is_a? self
18    end
19
20    def has_compound_domain
21      self != compound_domain
22    end
23
24    def method_added(m)
25      super
26    end
27
28    def singleton_method_added(m)
29      super
30    end
31
32    make_compound_domain :+
33    make_compound_domain :-
34    make_compound_domain :&
35
36    alias :union :+
37    alias :intersect :&
38    alias :difference :-
39  end
40
41  def part_of?(x)
42    if x.singleton_methods.include? :value?
43      x.value? self
44    end
45
46    raise TypeError.new("#{x} is not a domain.")
47  end
48
49  def value
50    return self

```

```

51     end
52 end

```

Listing 9: Monkey patching Object class

Finally, the domain adds new methods to Object class so that all class behave similarly to the domains, as type is a subset of domain in type checking sense. This allows classes that already exist, such as Integer and String class, act similarly to the domain, which allows the program to treat them like one for various benefits such as the combination of domains.

Through these methods, the users of this library can create domain using syntax like this:

```

1 def is_integer(x)
2   return Integer(x) rescue false
3 end
4
5 domain :Int do
6   rule(Integer)
7   rule(String) { |x| is_integer?(x) }
8 end

```

Listing 10: Sample code: declaration of domains

The domain `:Int do ...end` defines domain `Int` with two rules. The first rule states that it can accept any Integer, and the second rule states that if the object is String, domain `Int` can accept values that returns true for the `is_integer?(x)` method, which checks if the `x` is an integer. With this domain, it is possible to use both integer and string only containing integer as if they're integer, which can be a very power tool.

### 4.3 Assignment of signatures to function/methods

Unlike the declaration of domain, the implementation of signatures require heavy exploitation of Ruby's metaprogramming features, which can be difficult to understand. Additionally, the implementation is very long, so it requires extensive explanation on how they are implemented.

The signature is implemented in three steps. First, the signature reads the String passed in as a signature and parse and interpret the string to see if the String is a valid signature. The string is then transformed into two arrays, each indicating the rules for argument and rules for return values respectively. The arrays are then used to wrap the new method with checks that makes sure the argument and return value conforms to the rules and the original method is overridden with the new wrapped method. We will discuss each parts in detail over this section

```

1 def parse_tokens(sig)
2   valid_tokens = ['(', ')', ',', '[', ']', '$', '{', '}', '&']
3   sig.gsub!(/\s/, "")
4
5   tokens = []
6
7   other = ""
8   token = ""
9
10  state = :default
11  redo_state = false
12
13  sig.each_char do |x|
14    case state
15    when :default
16      token += x
17      case x
18      when '*'

```

```

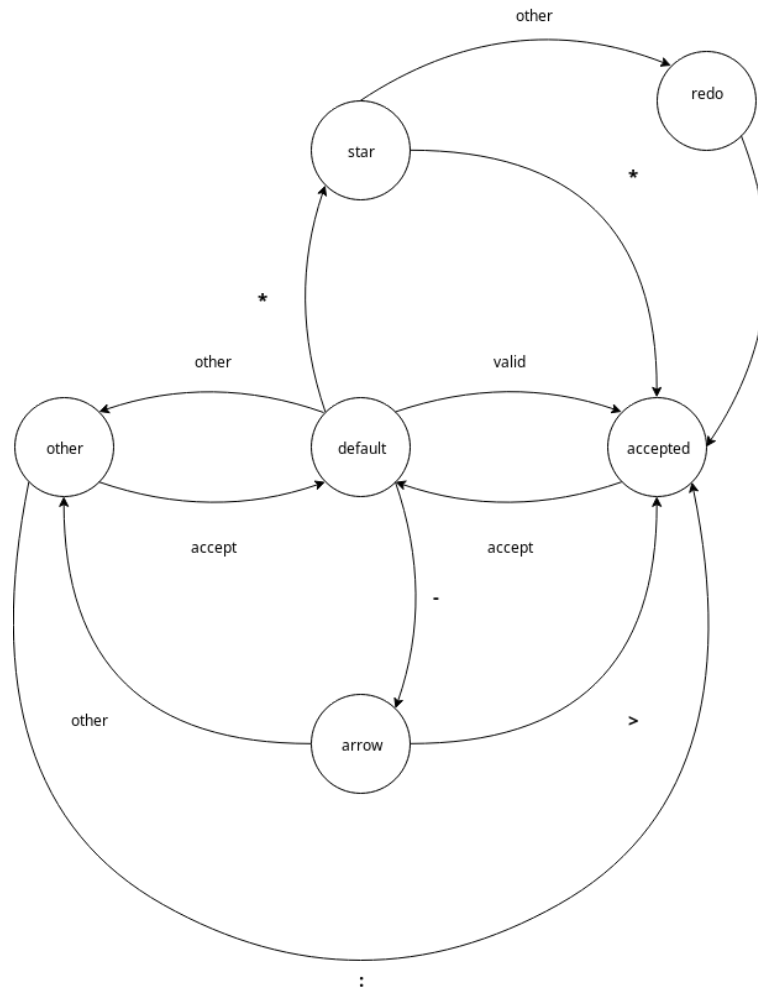
19         state = :star
20     when '-'
21         state = :arrow
22     when *valid_tokens
23         state = :accepted
24     else
25         state = :other
26     end
27 when :star
28     case x
29     when '*'
30         state = :accepted
31         token += x
32     else
33         state = :accepted
34         redo_state = true
35     end
36 when :arrow
37     token += x
38     case x
39     when '>'
40         state = :accepted
41     else
42         state = :other
43     end
44 end
45
46 if state == :other
47     other += token
48
49     if token == ":" then state = :accepted else state = :default end
50     token = ""
51 end
52
53 if state == :accepted
54     if !other.empty? then tokens << other; other = "" end
55     tokens << token if !token.empty?
56     token = ""
57     state = :default
58 end
59
60 if redo_state
61     state = :default
62     redo_state = false
63     redo
64 end
65 end
66
67 tokens << other if !other.empty?
68 tokens
69 end

```

Listing 11: Parser method

First, this method parses the string into list of tokens that can be used to interpret what it means. The method follows the following finite state machine to parse the string:

Figure 1: Finite State Machine



The machine begins at default, and moves to star state when it finds a \* symbol, arrow state (->) when it finds the - symbol, accepted state when it finds any of the valid tokens defined in the array, and other state for any other symbols. On star state, it moves to accepted state when it finds another \* symbol and redo state otherwise, which is a state that quickly moves to accepted state and re-examine the current symbol. The arrow state changes to accepted state only when it finds the >symbol which completes the arrow. The remaining other symbols move to other state. Other state simply concatenates the value to a variable, and if : was found, Other moves to accepted state to save it as a keyword. On accepted state, the tokens that has been found is stored in an array. After the entire string has been parsed, the array is returned.

```

1 def interpret_tokens(cl, local, tokens)
2   # Get tokens for arg and return separated
3
4   k = tokens.slice_after { |x| x == '->' }.to_a
5   a, r = k
6   a.pop
7   length = k.length
8
9   if length != 2
10     raise ArgumentError.new "Expected only one arrow (->), got #{length - 1}"
11   end
12
13   # Change the list of tokens into something more usable

```

```

14
15 # Ignore the () if it's properly at the beginning and end
16 a = a[1..-2] if a[0] == '(' && a[-1] == ')'
17 r = r[1..-2] if r[0] == '(' && r[-1] == ')'
18
19 a = retrieve_tokens(cl, local, a)
20 r = retrieve_tokens(cl, local, r)[0]
21
22 return a, r
23 end

```

Listing 12: Interpreter method

Now that all the tokens are in array, the next step is to store them into two arrays, one for argument and another for return values. Once the tokens are separated, it is sent to retrieve token method, which will turn all the tokens into something the wrapper can use to check the values.

```

1 def retrieve_tokens(cl, local, tokens)
2   # initialization
3
4   tokens.each do |x|
5     case state
6     when :token
7       case x
8       when '['
9         # array logic
10      when '{'
11        # hash logic
12      when ',', ']', '}', ''
13        # error
14      when '*', '**'
15        # star logic
16      when '$'
17        # optional logic
18      when '&'
19        # block logic
20      else
21        # non-token logic
22      when :comma
23        # comma logic
24      end
25    end
26    return arg, kwarg
27 end

```

Listing 13: Retriever method

The retrieve\_token method can be generalized through this code. After initializing important variables, the method iterates through all the tokens that are given, and performs different checks to see what the token means. There are two states in this method, which are “token” and “comma”. Because the tokens alternate between valid token and comma in real code, this two states are alternated between each other to make sure all tokens separated by comma means something.

```

1 def wrap_method(signature)
2   # initialize
3
4   return_trace = TracePoint.new(:return) do |tp|
5     # parse signature
6     # initialization for the new method
7     # check if the method has correct argument(s)
8     # wrap the method with checks and replace it
9     # add it to the correct place and with right scope
10  end

```

```

11
12   line_trace = TracePoint.trace(:line) do |tp|
13     # get a binding
14     # enable return_trace and disable line_trace
15   end
16 end

```

Listing 14: Wrapper method: overview

Next, the wrapper method uses the established parser in order to wrap the method correctly. Because this method is large, all logic will be separated into multiple sections to explain them all in detail.

```

1 return_trace = TracePoint.new(:return) do |tp|
2   # ...
3   # Check for validity of the method by getting its arity and aligning it with
4   # the length of arguments in signature
5   method = tp.self.instance_method(method_name) if tp.method_id == :method_added
6   method = tp.self.method(method_name) if tp.method_id == :
7   singleton_method_added
8
9   # Check if the parameters should have star variable (*arg) or double star
10  # variable (**arg)
11  has_star = false
12  has_dstar = false
13
14  args.each do |x|
15    has_star = true if is_star?(x) && x.star == '*'
16    has_dstar = true if is_star?(x) && x.star == '**'
17  end
18
19  has_dstar = !kwargs.empty? unless has_dstar
20
21  # Get all the parameters for the method
22  param = method.parameters
23
24  # initialization
25  optional_arg = []
26  optional_kwarg = []
27  expected_length_arg = 0
28  expected_length_kwarg = 0
29
30  # args is nil, so there should not be any parameters except for the blocks
31  if args.length == 1 && args[0].nil? && param.reject { |x| x[0] == :block }.
32  length == 0
33    next
34  else
35    # Then see if the variables are structured properly, such as making sure
36    # that star variable is a 3rd variable if the signature also have star variable
37    # for 3rd
38    param_arg = param.reject { |x| x[0] != :req && x[0] != :opt && x[0] != :
39    rest }
40    if param_arg.length != args.length
41      raise SignatureViolationError.new "Wrong number of total arguments:
42      Expected #{args.length}, found #{param_arg.length}"
43    end
44    # Iterate every rules
45    args.zip(param).each_with_index do |x, i|
46      ar, par = x
47
48      # Check if the parameter is what we expected
49      case
50      when ar.class == Class
51        if par[0] != :req
52          raise SignatureViolationError.new "Expected a required
53          variable for #{par[1]}, found #{par[0]}"

```

```

45         end
46         expected_length_arg += 1
47         when is_star?(ar)
48             if par[0] != :rest
49                 raise SignatureViolationError.new "Expected a * variable for
#{par[1]}, found #{par[0]}"
50             end
51             when is_optional?(ar)
52                 optional_arg << i
53                 if par[0] != :opt
54                     raise SignatureViolationError.new "Expected an optional
variable for #{par[1]}, found #{par[0]}"
55                 end
56                 expected_length_arg += 1
57             end
58         end
59
60         # Do the same thing for the keyword section
61         key_matched = []
62
63         param_kwarg = param.reject { |x| x[0] != :keyreq && x[0] != :key}
64         if param_kwarg.length != kwargs.length
65             raise SignatureViolationError.new "Wrong number of keyword arguments:
Expected #{kwargs.length}, found #{param_kwarg.length}"
66         end
67
68         param.each do |par|
69             if kwargs.has_key?(par[1])
70                 key_matched << par[1]
71                 if is_optional?(kwargs[par[1]])
72                     optional_kwarg << par[1]
73                 end
74             end
75         end
76
77         missing = kwargs.keys - key_matched
78         missing = [] if has_dstar
79
80         if !missing.empty?
81             raise SignatureViolationError.new "The following keywords are not in
the argument: #{missing}"
82         end
83     end
84     # ...
85 end

```

Listing 15: Wrapper method: check if the method has correct argument(s)

The second step in wrapping method, after the string have been parsed, is to determine if the method given has the right argument at the right area:

```

1 # error, first argument should be required, and second should be optional
2 domain 'String, $String -> nil'
3 def f(x = 5, y)
4 end
5
6 # no error
7 domain 'String, $String -> nil'
8 def f(x, y = 5)
9 end

```

Listing 16: Example: wrapper method

As evident by this example, the first method f does not follow the rules provided. The signature stated

that the first argument is required while second argument is optional, yet the `f` provides the opposite. Additionally, there can be too much or too little arguments on `f` that would break the rule. To prevent this type of rule breaking, this part of wrapper goes through all the parameters of the method and the rules for the argument and check if they match.

```

1 return_trace = TracePoint.new(:return) do |tp|
2   # ...
3   # wrap the method
4   lamb = lambda do
5     tp_self.send use, method_name do | *arg1, **arg2, &block |
6       old_arg1 = arg1
7       old_arg2 = arg2
8
9       all_args = [pad_with_optional(arg1, optional_arg, expected_length_arg)
10      , arg2]
11
12      arg_types = args
13      kwarg_types = kwargs
14
15      if has_star
16        check_array all_args[0], arg_types
17      else
18        (all_args[0].zip(arg_types)).each do |arg_type|
19          arg, type = arg_type
20          check_validity arg, type
21        end
22      end
23
24      check_keyword all_args[1], kwarg_types
25
26      if arg1.length == 0 && arg2.length == 0
27        r = self.send(old_method_name) { block.call } if !block.nil?
28        r = self.send(old_method_name) if block.nil?
29      elsif arg2.length == 0
30        r = self.send(old_method_name, *old_arg1) { block.call } if !block
31        .nil?
32        r = self.send(old_method_name, *old_arg1) if block.nil?
33      elsif arg1.length == 0
34        r = self.send(old_method_name, **old_arg2) { block.call } if !
35        block.nil?
36        r = self.send(old_method_name, **old_arg2) if block.nil?
37      else
38        r = self.send(old_method_name, *old_arg1, **old_arg2) { block.call
39        } if !block.nil?
40        r = self.send(old_method_name, *old_arg1, **old_arg2) if block.nil
41        ?
42      end
43
44      all_rets = if r.is_a?(Enumerable) then r else [r] end
45      ret_types = if ret.is_a?(Enumerable) then ret else [ret] end
46
47      all_rets.zip(ret_types).each do |ret_type|
48        ret, type = ret_type
49        check_validity ret, type
50      end
51
52      all_rets
53    end
54  end
55  # ...
56 end

```

Listing 17: wrap the method with checks and replace it



Next, the wrapper creates an anonymous function that overrides the original method with a new method that checks the validity of argument and return values. Once the method is wrapped, the method will always go through the checks before and after the method runs to make sure everything is accurate.

```

1 return_trace = TracePoint.new(:return) do |tp|
2   # ...
3   trace = TracePoint.trace(:line) do |tp|
4     tp.disable
5     # create the newly defined method in the appropriate places
6     if self.inspect == 'main'
7       Object.class_eval do
8         Object.alias_method(old_method_name, method_name)
9         lamb.call
10        private method_name, old_method_name
11      end
12    else
13      self.class_eval do
14        self.alias_method(old_method_name, method_name)
15        private old_method_name
16        pr = tp_self.private_methods(false).include?(method_name)
17        lamb.call
18        private method_name if pr
19      end
20    end
21  end
22 end

```

Listing 18: Wrapper method: Add it to the correct place with the right scope

Next, we make sure the method is added to the correct place. If the method is created on main, or top level, then internally, it is the same as creating a private method in Object class in Ruby, so it does exactly that. Otherwise, it should be in some kind of class, so it performs the action on the self variable, which should hold the class information. It is also important to figure out the encapsulation this method is intended to be, because the new method will always be public other wise. There are 3 ways to make the method private:

```

1 # 1. Everything after this private keyword is considered private
2 private
3 def f; end          # private
4 def g; end          # private
5
6 # 2. Appending private to the front makes only this method private
7 private def f; end  # private
8 def g; end          # not private
9
10 # 3. You can write the name of the function to privatize after the method has been
    created
11 def f; end
12 def g; end
13
14 private :f          # f() is now private

```

Listing 19: Private methods

Making methods private on 2 and 3 is trivial. 2 turns method into private right after the method is created, and since the wrap method activates right before the method is fully added, the private keyword turns the new, wrapped method into private instead. 3 turns the method into private the moment the “private :name” line runs, which is long after the new method has been replaced, so the new method turns into private. The first one, however, is a challenge. Although I could not figure out exactly when the function turns private, the method did not automatically turn private, which indicates that the private keyword transforms the method before it finishes adding the value, which the wrap method overrides into public. So my next step was to create a TracePoint for line that waits a single line. By waiting a single line,

the instruction pointer moved outside the scope of the wrapper function, which is after the old method has been properly added. The algorithm then checks if the old method was added as a private method, and turn it into private if it was the case. By doing so, I managed to account for all three different ways to make the method private.

```

1 return_trace = TracePoint.new(:return) do |tp|
2   # ...
3 end
4
5 line_trace = TracePoint.trace(:line) do |tp|
6   line += 1
7
8   if line == 1
9     local_binding = tp.binding
10  end
11
12  if line == 2
13    if tp.method_id != :method_added && tp.method_id != :singleton_method_added
14      raise NoMethodAddedError.new "No new method has been defined for
signature '#{signature}'"
15    end
16
17    return_trace.enable
18    tp.disable
19  end
20 end

```

Listing 20: The other TracePoints

Now that the basic flow of the wrap method is done, it is time to discuss why everything occurs within the return\_trace and line\_trace. In Ruby, there are methods called hook methods. Hook methods are basically methods that gets called when certain type of action gets performed. For example, method\_missing defined in the class is called whenever the program tries to call a method that does not exist. By overriding this method, it is possible to allow programs to respond to such command:

```

1 class T
2   def self.method_missing(m)
3     puts "#{m} was called"
4   end
5 end
6
7 a = T.new
8 a.test           #Test was called
9 a.new_method    #new_method was called

```

Listing 21: method\_missing example

It is a powerful tool to allow programmers to create complex programs that update itself. In my case, there is a relevant hook method named method\_added, that gets called after the method has been added. However, because anyone can override the method\_added method, it would be highly risky to override it in the library for the wrapping. Instead, I decided to use the tracepoints to wait for the method\_added to be ran, then insert its code at the end of the method\_added. This way, other programmers can override the method\_added method however they want to and TracePoint would always run the wrapping method regardless of the content.

line\_trace, on the other hand, makes sure the method\_added was called as soon as the signature was given. This is to prevent people from writing random codes between the signature and the method, so that it's obvious which method the signature is binding:

```

1 # ok
2 domain 'Integer -> String'
3 def f(x)

```

```

4     "hello"
5 end
6
7 # error, something is between method and signature
8 domain 'Integer -> nil'
9 x = 5
10 def g(x); end

```

Listing 22: Error example

Through these long string of codes, the programmer can write signatures for the function

```

1 # create a signature using domain method
2 domain 'Integer -> Integer'
3 def f(x)
4     5 * x
5 end
6
7 # It is also possible to use all 7 different types of arguments in Ruby without
8   any problem. &block variables are ignored in the signature
9 domain 'Integer, $Integer, *Integer, key1: Integer, key2: $Integer, **Integer ->
10   Integer'
11 def g(a, b = 1, *c, key1:, key2: 2, **key3, &block)
12     a + b
13 end
14
15 # There are two keywords in signature as of now, %any% and nil. %any% accepts any
16   value, while nil is when there are no argument or return value
17 domain '%any% -> nil'
18 def h(a); end

```

Listing 23: Signature demo

## 4.4 Declaration of variables

Simulating the declaration of variables is possible to some extent by creating an instance of the domain:

```

1
2 x = CreditCardNumber.new
3
4 x.value = "1000200030004000" # ok
5 x.value = "test"             # error

```

Listing 24: declaration of variables example

However, there are few minor problems that rises from doing such thing. One of them is that the value of x can easily be replaced if you assign another value without using the x.value command, and another is that although it is fine as a proof of concept, having to type .value everytime feels very tedious. So instead of having to do all that, the code should ideally look something like this:

```

1
2 CreditCardNumber :x
3
4 x = "1000200030004000" # ok
5 x = "test"             # error

```

Listing 25: declaration of variables example

The code looks simpler than the previous code, and it also resembles the declaration of variables in the statically typed languages. To imititate such feature, I made the define\_initializer

```

1 def define_initializer(a, domain)
2   define_method a do |sym|
3     layer = 1
4     bind = nil
5     obj_id = 8
6     line_checked = 0
7     path_checked = 0
8
9     line_trace = TracePoint.trace(:line) do |x|
10      if bind != nil
11        new_obj_id = 0
12        value = bind.local_variable_get(sym) if bind.
13        local_variable_defined?(sym)
14        new_obj_id = value.object_id if bind.local_variable_defined?(sym)
15
16        if new_obj_id != obj_id && line_checked != 0 && path_checked != 0
17          if !domain.value? value
18            raise ValueOutOfBoundsError.new "from #{path_checked}:#{
19              line_checked}: '#{value}' assigned to variable '#{sym}', which is out of bounds
20              from '#{domain.name}'"
21            end
22
23            val = domain.new
24            val.value = value
25
26            bind.local_variable_set(sym, val)
27          end
28
29          obj_id = new_obj_id
30
31          path_checked = x.path
32          line_checked = x.lineno
33        else
34          next
35        end
36      end
37
38      call_trace = TracePoint.trace(:call) do |x|
39        line_trace.disable if line_trace.enabled?
40
41        layer += 1
42      end
43
44      return_trace = TracePoint.trace(:return) do |x|
45        layer -= 1
46
47        if layer == 0
48          line_trace.enable
49        end
50
51        if layer == -1
52          line_trace.disable
53          x.disable
54          call_trace.disable
55        end
56      end
57
58      bind_trace = TracePoint.trace(:line) do |x|
59        bind = x.binding
60        bind_trace.disable
61      end
62    end
63  end
64 end

```

Listing 26: Method for defining the initializer method

This method creates a method with the same name as the domain that can initialize a variable. The method takes in the variable name you want to restrict, and create four TracePoints, each triggered at different point in code.

First, the `line_trace`. This TracePoint activates itself on every line to check if the variable has been changed. The variable check is done through checking the `object_id` of the variable. If the `object_id` is different from the previous line, then the new value assigned to the variable is checked using the domain the initializer is associated with. If the value passed the test, then the value is assigned to the variable without any trouble, and the `object_id` is updated. If the value did not pass the test, then the method raises an error stating that the value is outside of the domain.

Because checking every single line in the code will be a huge overhead, there must be ways to pause the TracePoint when it is not being used. The variable is typically not in use when another method gets called, so the `call_trace` and `return_trace` is there to control the `line_trace`. The `call_trace` will be activated whenever the program call a method. Because the calling the method moves the scope outside of where the initialized variable is effective, the `call_trace` disables the `line_trace` and increment the local variable layer by one. Once a method finish and the method returns, the `return_trace` will be activated. The `return_trace` will decrement the layer the `call_trace` incremented to indicate that it is one method closer to the original scope. If the layer becomes 0, then the scope has successfully returned to the original scope, so `line_trace` is enabled. If the layer reaches -1, then that means that the method this variable is defined on has finished. At this point, none of the TracePoints will be useful, so all three TracePoint will be disabled, and the line checking will not happen again.

`bind_trace` is a simple TracePoint that waits for the initializer method to end, then it grabs a binding from the scope the variable is from. This is to let `line_trace` figure out the status of the variable the initializer is tracking.

Because this can be difficult to visualize, consider the following code:

```
1 def f
2   CreditCardNumber :x
3
4   x = "1000200030004000"
5
6   bar
7
8   x = "5000600070008000"
9 end
10
11 def bar
12   x = 500
13   foo
14 end
15
16 def foo; end
```

Listing 27: Declaration of domain: demo

The code executes in the following order:

1. line 2: Creates the three tracepoints
2. line 4: Assign value to x
3. `line_trace`: Check the value of x, the value passes the test
4. line 6: run `bar()`
5. `call_trace`: Increment layer, disable `line_trace`, layer = 1
6. line 12: Assigns value to x. `line_trace` is disabled, so it is ignored
7. line 13: run `foo()`
8. `call_trace`: Increment layer, layer = 2

9. line 16: exit foo()
10. return\_trace: Decrement layer, layer = 1
11. line 14: exit bar()
12. return\_trace: Decrement layer, layer = 0, enable line\_trace because layer equals 0
13. line 8: Assign value to x
14. line\_trace: Check the value of x, the value passes test
15. line 9: exit f()
16. return\_trace: Decrement layer, layer = -1, disable all TracePoint because layer equals -1

## 4.5 Combination of domains

Combination of domains was simple feature to implement.

```

1 class CompoundDomain
2   def initialize(left, operand, right)
3     @left = left
4     @operand = operand
5     @right = right
6   end
7
8   def value?(x)
9     l = @left.value?(x)
10    r = @right.value?(x)
11
12    case @operand
13    when :+
14      return l || r
15    when :&
16      return l && r
17    when :-
18      return l && (not r)
19    else
20      raise ArgumentError.new "Invalid operation. #{@operand} is not a
valid operation"
21    end
22  end
23
24  def rules
25    return { left: @left.rules, operand: @operand, right: @right.rules }
26  end
27
28  def translators
29    return { left: @left.translators, operand: @operand, right: @right.
translators }
30  end
31 end

```

Listing 28: Compound Domain class

The domain that are created after the set operators are replaced with this CompoundDomain class. The CompoundDomain class overrides 3 methods, which are value?, rules, and translators. The value? method will execute the value? method for both left and right side of the operand to determine if the argument is valid for these two domains. Then, it performs boolean operation based on the operand that is passed in at the initialization process. Because CompoundDomain is shaped like a binary tree, recursively calling value? will traverse all checks that are part of the CompoundDomain's tree of rules, and successfully determine the correct answer. The rules and translators override is mostly for debugging process. It returns all rules or translators that are used by CompoundDomain.

## 4.6 Implicit conversion of variables

There are two instances of implicit conversions Ruby do, which are implicit conversion methods and coercing/try\_conversion. The implicit conversion methods are called by Ruby whenever the variable should act like another variable. For example, `to_str` is called automatically when you try to use non-string classes on something that expects a string value, such as interpolation. Coercing and `try_conversion` are also another way Ruby converts object into a more convenient object. Coerce is used in mathematical operations such as addition. Whenever Ruby finds a non-numerical value on the right side of the operation, it calls `coerce` method on the right side, which determines how Ruby should move forward with the operation. Similarly, `try_convert` is called when values need to be different type to be proceed. The implicit conversion method is called within the `try_convert` method.

```
1 def generate_translators
2   output = [default] + (translators.keys.map { |x| x[1] }) if !default.nil?
3   output = (translators.keys.map { |x| x[1] }) if default.nil?
4   output.uniq.each do |out|
5     converters = [Array, Hash, Integer, IO, Proc, Regexp, String]
6
7     if converters.include? out
8       a = "ary"      if out == Array
9       a = "hash"     if out == Hash
10      a = "int"       if out == Integer
11      a = "io"        if out == IO
12      a = "proc"      if out == Proc
13      a = "regexp"    if out == Regexp
14      a = "str"       if out == String
15
16      define_method "to_#{a}" do
17        if @value.class == self.class
18          return @value.send "to_#{a}"
19        end
20
21        val = self.class.send :translate, @value.class, out, @value
22
23        val
24      end
25    end
26
27    define_method "to_#{out.name}" do
28      if @value.class == self.class
29        return @value.send "to_#{out.name}"
30      end
31
32      val = self.class.send :translate, @value.class, out, @value
33
34      val
35    end
36  end
37
38  numerals = [Float, Integer]
39  numeral_output = numerals & output
40
41  define_method :coerce do |other|
42    return [other, self.value(numeral_output[0])] if !numeral_output.empty?
43
44    nil
45  end
46 end
```

Listing 29: Implicit conversion: implicit conversion methods

The `generate_translators` generate every implicit conversion methods the Ruby naturally support that

applies to the domain. The implicit conversion methods defined is based on the output types of every translations that are provided in the declaration of domain. Additionally, coerce method is defined based on possible numerical translations the domains have. If there are ways to translate into a numerical value, then coerce is defined based on such translations.

With this generator method, Ruby can convert implicitly on many occasions:

```

1 domain :Int do
2   rule(Integer)
3   rule(String) { |x| is_integer?(x) }
4
5   translation(Integer, String) { |x| x.to_s }
6   translation(String, Integer) { |x| Integer(x) }
7   translation(Integer, Array) { |x| [x, x, x] }
8 end
9
10
11 Int :x
12
13 x = "500"
14
15 puts 200 + x          # converts String to Integer implicitly
16 a, b, c = x          # converts String to Integer, then Integer to Array
17                       implicitly

```

Listing 30: Implicit conversion demo

```

1 def method_missing(name, *args, &blocks)
2   cl = self.class
3   output = [default] + (cl.translators.keys.map { |x| x[1] }) if !default.nil?
4   output = (cl.translators.keys.map { |x| x[1] }) if default.nil?
5
6   result = nil
7
8   binary_operations = %i[% & * ** + - / < << <= <=> == > >= >> ^]
9
10  output.each do |x|
11    # grab all methods it responds to
12    meth = x.methods
13
14    # If the output can respond to the value
15    if meth.include? name
16      if binary_operations.include? name
17        value = self.value(x)
18        begin
19          result = instance_eval("#{value} #{name} #{args[0]}")
20        rescue TypeError, ArgumentError
21          next
22        end
23      else
24        value = self.value(x)
25        begin
26          result = value.call(name, *args) { blocks.call } if !blocks.
27          nil?
28          result = value.call(name, *args) if blocks.nil?
29          rescue TypeError, ArgumentError
30            next
31          end
32        end
33      end
34    end
35
36    return result if !result.nil?
37
38    super

```



38 end

Listing 31: Implicit conversion: method\_missing

The next step is to allow implicit conversion based on what methods the output class can respond to. By overriding the method\_missing method, I made the domains respond to the missing methods. It checks if that method exists in any possible output translation. If it does, it translates the value into the correct value then tries to run that method. If the method returns error, then it continues searching. Otherwise, the result is returned as if the domain had that method all along. If it failed to find the method in any of the outputs, then error is raised as missing method.

```
1 def translate(d_in, d_out, value)
2   if d_in == d_out
3     return value
4   end
5
6   route = get_path(d_in, d_out)
7   route = fix_path(route)
8   val = value
9
10  route.each do |path|
11    val = translators[path].call(val) if (!translators[path].nil? && !val.nil?)
12  end
13
14  val
15 end
16
17 def get_path(src, dest)
18   checked = []
19   queue = [[src]]
20   found = false
21
22   while !queue.empty? && !found
23     head = queue[0][-1]
24     if head == dest
25       found = true
26     else
27       queue.shift
28     end
29
30     if !checked.include?(head) && !found
31       checked << head
32
33       # get all paths not checked
34       neighbors = @translation_map[head] - checked
35
36       if !neighbors.empty?
37         path = [head]
38         neighbors.each do |n|
39           queue << path + [n]
40         end
41       end
42     end
43   end
44
45   queue[0]
46 end
47
48 def fix_path(path)
49   prev = nil
50   curr = nil
51   fixed = []
52
```

```

53   path.each do |x|
54     if curr.nil?
55       curr = x
56     else
57       prev = curr
58       curr = x
59     end
60
61     if !prev.nil?
62       fixed << [prev, curr]
63     end
64   end
65
66   fixed
67 end

```

Listing 32: Implicit conversion: translation

Now that the implicit conversion algorithms are clear, we will examine how the values are translated. Notice in the first example given, the value translated from String to Array successfully, despite the lack of translation rules that directly link the two together. This is done by first mapping the translation rules into a graph. Using the breadth first search algorithm, the algorithm traverses the map to find the shortest path from source type to desired type. After the route has been figured out, the `fix_path` method takes the array of types list of input-output pairs. This array is then used to finally translate the value into the desired output.

## 4.7 Type Checking at Compile Time

Unfortunately, I could not implement the type checking system that runs on compile time during my time at Aoyama Gakuin University. There are several ideas I had about how to implement them, such as creating a brand new interpreter that checks the types before running, or using the extended hook method to run a check of the whole program before the script actually runs to imitate compile time checking, but both of the ideas require extensive programming to make something effective. Due to the two months time limit, I have decided to not implement them during this internship.

Although I was unable to implement this feature, I believe that the library does a very good job being an proof of concept for the domain to demonstrate how they work. Compile time type checking is passive feature that does not require the programmer's participation, while the remaining features are something programmers actively use, so the actual feel of the domain should be similar to how it would look like in a real life situation.