Refatorações para Transformação de Instruções Dinâmicas em Estáticas: Um Estudo em Ruby

Elder Rodrigues Jr, Ricardo Terra

Universidade Federal de Lavras, Lavras, Brasil

elderjr@computacao.ufla.br,terra@dcc.ufla.br

Abstract. *Dynamic features offered by programming languages allow to perform dynamic calls, dynamic constructions, and execute strings as codes. Such features provide programmers greater flexibility, ease development of frameworks, and reduction of duplicate codes. However, unnecessary uses of dynamic features jeopardize the code in several respects, such as readability, comprehension, and maintainability. Therefore, this work proposes and formalizes 20 refactorings to replace dynamic statements with static ones. We conducted an evaluation on 28 open-source Ruby systems where we could refactor 743 of 1,651 dynamic statements (45%).*

Resumo. *Instruções dinâmicas presentes em linguagens de programação permitem realizar chamadas dinâmicas, construções dinâmicas e execução de strings como código. Tais instruções oferecem ao programador uma maior flexibilidade, facilidade no desenvolvimento de frameworks e redução de trechos de código duplicados. No entanto, o uso desnecessário das instruções dinâmicas prejudica o código em diversos aspectos como na legibilidade, compreensão e manutenibilidade de software. Portanto, este trabalho propõe e formaliza 20 refatorações que substituem instruções dinâmicas por estáticas. Em uma avaliação em 28 sistemas Ruby de código aberto, foram refatoradas 743 de 1.651 instruções dinâmicas (45%).*

1. Introdução

Linguagens de programação usualmente oferecem instruções dinâmicas parametrizáveis que permitem realizar chamadas dinâmicas, construções dinâmicas e execuções de `strings` como código. O uso de tais funções – que ocorre mais frequentemente em linguagens dinamicamente tipadas – oferecem ao desenvolvedor vantagens como redução de trechos de código duplicados, flexibilidade e maior facilidade para o desenvolvimento de *frameworks* [Rodrigues and Terra 2017a]. Por outro lado, o *problema* ocorre no uso desnecessário das instruções dinâmicas que pode prejudicar: (i) abordagens de análises estáticas [Callaú et al. 2011, Callaú et al. 2013, Hills 2015, Holkner and Harland 2009]; (ii) legibilidade, manutenibilidade, complexidade e compreensão do código [Rodrigues and Terra 2017a]; (iii) detecção antecipadas de erros, o que esconde erros de tipos [Ren et al. 2013, Furr et al. 2009b] ou violações arquiteturais [Miranda et al. 2016a]; (iv) otimizações feitas por compiladores [Callaú et al. 2011, Callaú et al. 2013]; e (v) funcionalidades de IDEs tais como auto-completar e refatorações [Rodrigues and Terra 2017a].

Diante disso, este artigo propõe e formaliza 20 refatorações para transformar instruções dinâmicas em estáticas. Um sistema de recomendação, denominado `nodyna`, implementa as refatorações propostas para a linguagem Ruby.[1] Primeiramente, aplica-se um algoritmo estático de inferência de valores (e de tipos, caso a linguagem alvo seja dinamicamente tipada). O objetivo é mapear todas as variáveis do sistema em triplas $[\texttt{m}, \{\texttt{String}\}, \{'\texttt{foo}'\}]$ que indica que a variável m pode assumir o tipo `String` e o valor $'$`foo`$'$. Analisa-se, então, as instruções dinâmicas a fim de identificar oportunidades de refatoração. Em uma avaliação em 28 sistemas Ruby de código aberto, `nodyna` sugeriu refatorações para 743 de 1.651 instruções dinâmicas (45%).

É imprescindível mencionar que as refatorações propostas neste artigo têm como *objetivo principal* expor o comportamento de instruções dinâmicas, melhorando a compreensibilidade, manutenibilidade e desempenho. No entanto, como instruções dinâmicas usualmente atuam com alto grau de generalidade, a maioria das refatorações propostas neste artigo aumentam a verbosidade pela adição de código, o que é de fato uma consequência inevitável ao objetivo principal.

O restante deste artigo está organizado como a seguir. A Seção 2 apresenta a linguagem Ruby e descreve as instruções dinâmicas abordadas neste artigo. A Seção 3 introduz o conceito de inferência de valores e tipos. A Seção 4 formaliza as refatorações propostas. A Seção 5 reporta uma avaliação em 28 sistemas Ruby de código aberto. Por fim, a Seção 6 discute os trabalhos relacionados e a Seção 7 conclui.

2. Background

2.1. Ruby

Ruby é uma linguagem relativamente nova, sendo criada em 1995 por Yukihiro Matsumoto. Tal linguagem possui diversas instruções equivalentes para permitir que o programador escreva o programa da forma mais natural possível [Furr et al. 2009a]. Devido a isso, a linguagem é conhecida por sua alta legibilidade e códigos limpos. Todo programa Ruby é desenvolvido por meio de objetos, uma vez que a linguagem é totalmente orientada a objetos [Black 2009]. Além disso, Ruby possui tipagem dinâmica e forte, e também é multiplataforma, i.e., é suportada por diversos sistemas operacionais. O Código 1 apresenta um exemplo da linguagem demonstrando a criação de classes, conceitos de herança e outras características.

É possível observar cinco principais características da linguagem Ruby: (i) criação do módulo `ModuleA` com um método `say_hi` (linhas 1-5); (ii) inclusão do módulo `ModuleA` na classe `ClassA` (linha 7), i.e., os métodos definidos no módulo são copiados para a classe; (iii) `ClassB` herda a classe `ClassA` (linha 12); (iv) instanciação de um objeto da classe `ClassB` (linha 17); e (v) invocações dos métodos definidos nas classes e módulo do código (linhas 18-20).

2.2. Instruções dinâmicas em Ruby

A linguagem Ruby possui algumas funções que permitem realizar, de forma dinâmica, invocações de métodos, criação de estruturas (classes, métodos, blocos, variáveis e constantes) e execução de *strings* como expressões. Tais funções são chamadas de instruções

[1] Embora o foco na linguagem Ruby, este estudo pode ser diretamente adaptado para outras linguagens.

```ruby
module ModuleA
  def say_hi
    puts "Hi"
  end
end
class ClassA
  include ModuleA
  def say_hello
    puts "Hello"
  end
end
class ClassB < ClassA
  def say_bye
    puts "Bye"
  end
end
obj = ClassB.new
obj.say_hello #Hello
obj.say_hi #Hi
obj.say_bye #Bye
```

Código 1. Exemplo de código Ruby

dinâmicas e permitem que ocorra mais agilidade no desenvolvimento do programa, evitando, por exemplo, que parte do código seja escrita manualmente (metaprogramação). Contudo, o uso de instruções dinâmicas pode, principalmente, prejudicar a complexidade de análise do código e, portanto, dificultar a manutenibilidade.

Este estudo analisa sete instruções dinâmicas. A seleção foi feita com base em um estudo anterior [Rodrigues and Terra 2017b] em que 12 instruções dinâmicas foram analisadas manualmente. No entanto, foram excluídas deste estudo apenas cinco instruções consideradas intrinsecamente complexas. Por exemplo, a função `eval` interpreta uma *string* como um código Ruby e, portanto, a tarefa de remoção de tal instrução seria de alta complexidade. Portanto, as sete instruções dinâmicas analisadas neste artigo são descritas a seguir:

1. `class_variable_set`: cria ou altera o valor de uma variável de classe (variável estática) dinamicamente.
2. `class_variable_get`: obtém o valor de uma variável de classe dinamicamente.
3. `instance_variable_set`: cria ou altera o valor de uma variável de instância dinamicamente.
4. `instance_variable_get`: obtém o valor de uma variável de instância dinamicamente.
5. `const_get`: obtém uma constante dinamicamente.
6. `const_set`: define uma constante dinamicamente.
7. `send`: invoca um método dinamicamente.

O Código 2 ilustra um exemplo que inclui as sete instruções dinâmicas mencionadas. Em poucas palavras, `const_set` define a constante `CONST` (linha 2), `const_get` obtém o valor da constante `CONST` (linha 8); `class_variable_set` define a variável de classe `staticVar` (linha 9); `class_variable_get` obtém o valor da variável de classe `staticVar` (linha 10); `instance_variable_set` define a variável de instância `instanceVar` (linha 11); `instance_variable_get` obtém o valor da variável de instância `instanceVar` (linha 12); e `send` invoca o método `say_hi` (linha 13).

```ruby
class ClassA
  const_set(:CONST, "VAL")
  def say_hi
    return "hi"
  end
end
obj = ClassA.new
ClassA.const_get(:CONST)
ClassA.class_variable_set(:staticVar, 1)
puts ClassA.class_variable_get(:staticVar) #1
obj.instance_variable_set(:instanceVar, 2)
puts obj.instance_variable_get(:instanceVar) #2
puts obj.send(:say_hi) #hi
```

Código 2. Exemplo de instruções dinâmicas em Ruby

3. Inferência de valores

O objetivo de uma inferência de valores é obter os valores que cada variável pode assumir. Em um contexto prático, o objetivo é mapear todas as variáveis do sistema em triplas $[\texttt{var}, \{\texttt{T}_1, \texttt{T}_2\}, \{\texttt{v}_1, \texttt{v}_2\}]$ que indica que a variável var pode assumir os tipos $\texttt{T}_1$ ou $\texttt{T}_2$ com os valores $\texttt{v}_1$ ou $\texttt{v}_2$.

Em linguagens dinamicamente tipadas, é importante que a inferência de valores trabalhe em conjunto com uma inferência de tipos. Portanto, como este estudo avalia as refatorações propostas em Ruby (uma linguagem dinamicamente tipada em que desenvolvedores largamente usam instruções dinâmicas), o algoritmo de inferência de valores foi implementado sob um algoritmo de inferência de tipos baseado em análise estática [Miranda et al. 2016b]. Basicamente, a partir de atribuições diretas de valores existentes no código fonte, o algoritmo propaga tais atribuições, inferindo valores de outras atribuições. O Código 3 ilustra o algoritmo de inferência implementado.

```ruby
class ClassA
  def foo(x)
    if(x.count > 5)
      return "str"
    else
      return 2
    end
  end
end
```

```ruby
class ClassB
  def bar
    a = ClassA.new
    str = "stx"
    return a.foo(str)
  end
end
```

Código 3. Inferência de valores

Inicialmente, apenas as atribuições diretas de valores são consideradas. Dessa forma, ocorre as seguintes inferências:

$$[\texttt{ClassA::foo\#return}, \{\texttt{String}, \texttt{Fixnum}\}, \{'\texttt{str}', 2\}] \quad (1)$$

$$[\texttt{ClassB::bar\#a}, \{\texttt{ClassA}\}, \{\texttt{ClassA}\}] \quad (2)$$

$$[\texttt{ClassB::bar\#str}, \{\texttt{String}\}, \{'\texttt{stx}'\}] \quad (3)$$

(1) o método `foo` retorna os tipos `String` com valor `str` e `Fixnum` com valor 2 (linhas 4 e 6 à esquerda); (2) a variável `a` recebe uma instância do tipo `ClassA` (linha 3 à direita); e (3) variável `str` do tipo `String` com valor `stx` (linha 4 à direita). Como

ocorreram novas inferências, o algoritmo irá propagá-las e, portanto, as seguintes novas inferências serão feitas:

$$[\texttt{ClassA::foo\#x}, \{\texttt{String}\}, \{'\texttt{stx}'\}] \quad (4)$$

$$[\texttt{ClassB::bar\#return}, \{\texttt{String}, \texttt{Fixnum}\}, \{'\texttt{str}', 2\}] \quad (5)$$

(4) o parâmetro formal `x` de `foo` recebe a variável `str` (linha 5 à direita), logo pode assumir os valores de `str` (ver Equação 3); e (5) o método `bar` retorna os valores retornados por `foo` (linha 5 à direita), logo pode assumir os valores de retorno de `foo` (ver Equação 1). Como não há estruturas que dependem das novas inferências produzidas, o algoritmo conclui.

É importante, todavia, mencionar que a atual implementação da heurística não é capaz de inferir tipos quando possui valores em estruturas e valores obtidos de forma dinâmica, além de poder gerar falsos positivos por não analisar o fluxo de execução.

4. Refatorações para transformação de instruções dinâmicas em estáticas

Com base nas triplas obtidas pela inferência de valores (Seção 3), é possível inferir quais os valores que estão sendo utilizados em instruções dinâmicas. Este artigo, portanto, propõe um conjunto de refatorações para converter tais instruções em códigos estáticos.

Conforme previamente esclarecido, as refatorações propostas neste artigo têm como *objetivo principal* expor o comportamento de instruções dinâmicas, melhorando a compreensibilidade, manutenibilidade e desempenho. No entanto, como instruções dinâmicas usualmente atuam com alto grau de generalidade, a maioria das refatorações propostas neste artigo aumentam a verbosidade pela adição de código, o que é de fato uma consequência inevitável ao objetivo principal.

Mais importante, devido à possibilidade de as instruções dinâmicas receberem valores externos (entradas de usuário, arquivos, respostas de microsserviços, etc.), a maioria das refatorações propostas inserem um bloco `else` que invoca a instrução dinâmica original, i.e., ela não é de fato excluída. Embora alguns podem atribuir isso a um *bad smell* de duplicidade de código, essa é a única forma de preservar o comportamento do código. No entanto, caso o desenvolvedor garanta a completude das refatorações sugeridas, o bloco *else* pode ser removido.

4.1. Formalização

Esta seção descreve, formaliza e ilustra cada uma das refatorações propostas. A Tabela 1 descreve as funções auxiliares utilizadas na formalização, tal como retornar os valores que uma certa variável pode assumir.

Tabela 1. Funções auxiliares

Função	Descrição
isStatic(var)	Verifica se o valor da variável `var` é estático
isDynamic(var)	Verifica se o valor da variável `var` é dinâmico
valuesOf(var)	Retorna os possíveis valores que a variável `var` pode assumir

4.1.1. Refatorações para `class_variable_set`, `class_variable_get`, `instance_variable_set` e `instance_variable_get`

A Tabela 2 apresenta as recomendações para o uso das instruções `class_variable_get`, `class_variable_set`, `instance_variable_set` e `instance_variable_get`. Basicamente, uma sugestão é dada para uma instrução dinâmica que se enquadre no *template* e satisfaça as *pré-condições* de uma refatoração. A primeira refatoração da Tabela 2, por exemplo, apresenta uma conversão trivial da instrução `instance_variable_get` em que um valor estático é recebido no primeiro parâmetro da instrução (`instance_variable_get('foo')`) e, portanto, é recomendado que a variável seja obtida diretamente (`@foo`). Um exemplo mais elaborado que apresenta as principais refatorações relacionadas às instruções mencionadas é ilustrado nos Códigos 4 e 5. Apenas foi ilustrado as refatorações aplicadas às instruções dinâmicas relacionadas a variáveis de instância, pois as refatorações relacionadas a variáveis estáticas são equivalentes.

Tabela 2. Refatorações de `class/instance_variable_set/get`

Ref.	*Template*	Pré-condições	Formalização
#1	`instance_variable_get(x)` `class_variable_get(x)`	`isStatic(x)`	x
#2	`obj.instance_variable_get(x)` `obj.class_variable_get(x)`	`isStatic(x)`	obj.getX()

#3	`instance_variable_get(x)`
	`class_variable_get(x)`
		`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then v`
			`else instance/class_variable_get(x)`

#4	`obj.instance_variable_get(x)`
	`obj.class_variable_get(x)`
		`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then obj.getV()`
			`else obj.instance/class_variable_get(x)`

#5	`instance_variable_set(x, y)` `class_variable_set(x, y)`	`isStatic(x)`	x = y
#6	`obj.instance_variable_set(x, y)` `obj.class_variable_set(x, y)`	`isStatic(x)`	obj.setX(y)

#7	`instance_variable_set(x, y)`
	`class_variable_set(x, y)`
		`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then v = y`
			`else instance/class_variable_set(x)`

#8	`obj.instance_variable_set(x, y)`
	`obj.class_variable_set(x, y)`
		`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then obj.setV(y)`
			`else obj.instance/class_variable_set(x)`

No Código 4, podem ser observadas duas invocações das instruções `instance_variable_set` e `instance_variable_get`. No primeiro caso (linha 3), o valor `val` é atribuído dinamicamente na variável de instância `@instanceVar`. Devido ao primeiro parâmetro da instrução ser estático ("`@instanceVar`") recomenda-se, portanto, realizar a atribuição sem utilizar a instrução dinâmica (quinta refatoração), conforme ilustrado na linha 3 do Código 5. No segundo caso (linha 9), a instrução `instance_variable_get` é utilizada para obter uma variável de instância nomeada pela variável `x`. Por isso, é recomendada a criação de estruturas condicionais para todos os possíveis valores de tal variável (terceira refatoração), conforme ilustrado nas linhas 8-14 do Código 5. A condição `else` (linhas 12-14 do Código 5) preserva o comportamento.

```
1  class ClassA
2   def qux()
3    instance_variable_set(
4     "@instanceVar1", val)
5    x = "@instanceVar1"
6    ...
7    x = "@instanceVar2"
8    ...
9    v = instance_variable_get(x)
10  end
11 end
```

Código 4. Usos das instruções `instance_variable_set/get`

```
1  class ClassA
2   def qux()
3    @instanceVar1 = val
4    x = "@instanceVar1"
5    ...
6    x = "@instanceVar2"
7    ...
8    if(x == "@instanceVar1")
9     v = @instanceVar1
10   elsif(x == "@instanceVar2")
11    v = @instanceVar2
12   else
13    v = instance_variable_get(x)
14   end
15  end
16 end
```

Código 5. Refatorações das instruções `instance_variable_set/get`

4.1.2. Refatorações para `send`

A Tabela 3 apresenta as refatorações para o uso da instrução `send`. A primeira refatoração, por exemplo, apresenta a refatoração trivial de `send` em que o primeiro parâmetro da instrução é um valor estático ("`foo`") e, portanto, recomenda-se a invocação do método diretamente (`foo()`). Um exemplo mais elaborado que apresenta as principais refatorações relacionadas à instrução mencionada é ilustrado nos Códigos 6 e 7.

Tabela 3. Refatorações de `send`

Ref.	*Template*	Pré-condições	Formalização
#1	`send(x)`	`isStatic(x)`	`x()`
#2	`obj.send(x)`	`isStatic(x)`	`obj.x()`
#3	`send(x)`	`isDynamic(x)`	$(\forall$ `v` $\in$ `valuesOf(x)`) `if(x == v) then v()` `else send(x)`
#4	`obj.send(x)`	`isDynamic(x)`	$(\forall$ `v` $\in$ `valuesOf(x)`) `if(x == v) then obj.v()` `else obj.send(x)`

No código 6, podem ser observadas duas invocações da instrução dinâmica. A primeira (linha 2), invoca o método `foo` dinamicamente através de um valor estático. Recomenda-se, portanto, invocar a função normalmente (primeira refatoração), como ilustrado na linha 2 do Código 7. Já no segundo caso (linha 9), o método a ser invocado é nomeado pelo valor da variável `x`. Assim, é recomendada a criação de estruturas condicionais para todos os possíveis valores de tal variável (terceira refatoração), como observado nas linhas 9–15 do Código 7. A condição `else` (linhas 13–15 do Código 7) preserva o comportamento.

```
1  def qux()
2    send("foo")
3  end
4
5  def qux()
6    x = "foo"
7    ...
8    x = "baa"
9    ...
10   send(x)
11 end
```

Código 6. Usos de `send`

```
1  def qux()
2    foo()
3  end
4
5  def qux()
6    x = "foo"
7    ...
8    x = "baa"
9    ...
10   if(x == "foo")
11     foo()
12   elsif(x == "baa")
13     baa()
14   else
15     send(x)
16   end
17 end
```

Código 7. Refatorações de `send`

4.1.3. Refatorações para `const_set` e `const_get`

A Tabela 4 apresenta as refatorações para as instruções `const_set` e `const_get`. A primeira refatoração, por exemplo, apresenta a refatoração trivial de `const_get` em que o primeiro parâmetro da instrução é estático ("`CONST`") e, portanto, recomenda-se a obtenção da constante diretamente (`CONST`). Um exemplo mais elaborado que apresenta as principais refatorações relacionadas às instruções mencionadas é ilustrado nos Códigos 8 e 9.

Tabela 4. Refatorações de const_set/get

Ref.	*Template*	Pré-condições	Formalização
#1	`const_get(x)`	`isStatic(x)`	x
#2	`obj.const_get(x)`	`isStatic(x)`	obj::x

#3	`const_get(x)`	`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then v`
			`else const_get(x)`

#4	`obj.const_get(x)`	`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then obj::v`
			`else obj.const_get(x)`

#5	`const_set(x, y)`	`isStatic(x)`	x = y
#6	`obj.const_set(x, y)`	`isStatic(x)`	obj::x = y

#7	`const_set(x, y)`	`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then v = y`
			`else const_set(x, y)`

#8	`obj.const_set(x, y)`	`isDynamic(x)`	(∀v ∈ `valuesOf(x)`)
			`if(x == v) then obj::v = y`
			`else obj.const_set(x, y)`

No Código 8, podem ser observadas duas invocações da instrução `const_set` e uma da instrução `const_get`. Na primeira (linha 2), a instrução `const_set` é utilizada para criar a constante `CONST1` dinamicamente, que é nomeada por um valor estático. Por isso, recomenda-se criar a constante diretamente, sem a utilização da instrução dinâmica (quinta refatoração), como ilustrado no Código 9 (linha 2). A segunda invocação de `const_set` (linha 4) é semelhante a primeira mencionada, porém, a instrução dinâmica é invocada pelo objeto `ClassA` e, portanto, recomenda-se criar a constante na forma `ClassA` :: `CONST2` (sexta refatoração), conforme demonstrado no Código 9 na linha 4. Por último, a instrução `const_get` é utilizada para obter uma constante dinamicamente

nomeada pela variável x (linha 10) e, portanto, recomenda-se a criação de estruturas condicionais para todos os possíveis valores de tal variável (terceira refatoração), conforme ilustrado nas linhas 9-15 do Código 9. A condição else (linhas 13-15 do Código 9) preserva o comportamento.

```
1  class ClassA
2   const_set("CONST1", 1)
3   def qux()
4    ClassA.const_set(
5      "CONST2", 2)
6    x = "CONST1"
7    ...
8    x = "CONST2"
9    ...
10   y = ClassA.const_get(x)
11   end
12 end
```

Código 8. Usos de const_set/get

```
1  class ClassA
2    CONST1 = 1
3    def qux()
4      ClassA::CONST2 = 2
5      x = "CONST1"
6      ...
7      x = "CONST2"
8      ...
9      if(x == "CONST1")
10       y = ClassA::CONST1
11     elsif(x == "CONST2")
12       y = ClassA::CONST2
13     else
14       y = ClassA.const_get(x)
15     end
16   end
17 end
```

Código 9. Refatorações de const_set/get

4.1.4. Limitações

As instruções dinâmicas apresentadas podem violar as regras de visibilidade, e.g., o método send pode invocar até mesmo métodos que são privados de outras classes. Em vista disso, caso a visibilidade de um método, constante ou atributo for violada, também é recomendado a alteração da visibilidade para pública. Entretanto, em alguns casos, a análise estática pode não reconhecer a declaração de métodos, constantes ou atributos (criações dinâmicas, por exemplo). Devido a isso, pode não ser possível verificar a visibilidade de tais estruturas. Portanto, nos casos em que a declaração de uma estrutura não é encontrada, é informado ao desenvolvedor que pode ser necessária a alteração da visibilidade.

4.2. Implementação de referência

Para demonstrar a aplicabilidade das refatorações propostas, um protótipo de um sistema de recomendação, denominado nodyna, implementa as refatorações propostas para a linguagem Ruby. Além de implementar um algoritmo estático de inferência de valores, como Ruby é dinamicamente tipada, adaptou-se uma inferência de tipos proposta em um estudo anterior [Miranda et al. 2016b]. Basicamente, nodyna analisa instruções dinâmicas para identificar oportunidades de uma das refatorações propostas. A implementação e seu código fonte estão publicamente disponíveis em:

`http://github.com/rterrabh/nodyna`

Devido à complexidade, a atual implementação de nodyna não lida com estruturas dinâmicas e análise de fluxo. Como exemplo da primeira limitação, não são analisados métodos criados dinamicamente. Como exemplo da segunda limitação, instruções inalcançáveis também são analisadas. No entanto, uma análise de fluxo poderia garantir se uma instrução é alcançável ou não, evitando falsos positivos. Outra limitação da ferramenta é em relação a características da linguagem Ruby. Execuções de blocos de instruções não são interpretadas, i.e., não serão feitas inferências em relação ao retorno de um

bloco e também às variáveis que tais blocos podem receber como parâmetro. Por último, herança e polimorfismo também não são considerados no algoritmo de inferência. Isso ocorre pois, na linguagem Ruby, a inclusão de um módulo em uma classe pode também ser feita dinamicamente, o que torna a análise complexa.

5. Avaliação

Esta seção avalia a aplicabilidade das refatorações propostas em sistemas reais desenvolvidos na linguagem Ruby. O critério de seleção foram os 28 sistemas Ruby mais populares do repositório `GitHub` em setembro de 2015. Além disso, tais sistemas já foram analisados manualmente em trabalhos anteriores [Rodrigues and Terra 2017b, Rodrigues and Terra 2017a]. Assim, executou-se `nodyna` em cada um dos sistemas cujo resultado é reportado na Tabela 5.

Tabela 5. Total de Refatorações sugeridas por nodyna

	class_variable_set	**class_variable_get**	**const_set**	**const_get**	**instance_variable_set**	**instance_variable_get**	**send**
Active Admin	0 / 0	0 / 0	0 / 1	0 / 2	0 / 1	0 / 2	20 / 36
Cancan	0 / 0	0 / 0	0 / 0	0 / 0	1 / 3	0 / 4	9 / 22
Capistrano	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	1 / 3
Capybara	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	9 / 10
Carrierwave	0 / 0	0 / 0	0 / 2	0 / 0	1 / 1	0 / 0	1 / 11
CocoaPods	0 / 0	0 / 0	0 / 0	0 / 1	0 / 1	0 / 0	11 / 20
Devdocs	0 / 0	0 / 0	0 / 0	0 / 3	1 / 1	0 / 0	20 / 21
Devise	0 / 0	0 / 0	0 / 0	1 / 4	0 / 5	2 / 3	4 / 27
Diaspora	0 / 0	0 / 0	0 / 0	0 / 1	15 / 17	11 / 13	36 / 50
Discourse	0 / 0	0 / 0	0 / 0	0 / 2	1 / 4	1 / 1	52 / 152
FPM	0 / 0	0 / 0	0 / 0	0 / 0	1 / 1	1 / 1	2 / 12
GitLab	1 / 1	0 / 0	0 / 1	0 / 3	0 / 0	1 / 1	20 / 50
Grape	0 / 0	0 / 0	1 / 1	2 / 2	0 / 0	0 / 0	5 / 8
Homebrew	0 / 0	0 / 0	1 / 2	1 / 3	1 / 1	0 / 0	16 / 39
Homebrew-Cask	0 / 0	0 / 0	0 / 0	0 / 5	1 / 1	0 / 0	3 / 15
Huginn	0 / 0	0 / 0	1 / 5	0 / 5	0 / 1	0 / 0	16 / 21
Jekyll	0 / 0	0 / 0	0 / 0	0 / 1	0 / 0	0 / 0	4 / 6
Octopress	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0	0 / 0
Paperclip	0 / 0	0 / 0	0 / 0	0 / 5	0 / 1	0 / 1	28 / 53
Rails	0 / 1	0 / 0	0 / 11	0 / 24	13 / 20	10 / 20	102 / 241
Rails Admin	0 / 0	0 / 0	0 / 0	0 / 1	5 / 9	2 / 9	9 / 50
Resque	0 / 0	0 / 0	0 / 0	0 / 2	0 / 0	1 / 1	1 / 10
Ruby	0 / 0	0 / 0	36 / 63	2 / 35	51 / 76	12 / 27	61 / 111
Sass	0 / 0	0 / 0	0 / 1	0 / 4	15 / 15	2 / 2	25 / 41
Simple Form	0 / 1	0 / 1	0 / 0	0 / 1	0 / 0	0 / 0	1 / 13
Spree	0 / 0	0 / 0	0 / 0	0 / 1	0 / 8	2 / 6	81 / 154
Vagrant	0 / 0	0 / 0	0 / 0	0 / 1	1 / 3	2 / 6	4 / 9
Whenever	0 / 0	0 / 0	0 / 0	0 / 0	0 / 1	0 / 0	2 / 2
Total	**1 / 3**	**0 / 1**	**39 / 87**	**6 / 106**	**107 / 170**	**47 / 97**	**543 / 1187**

Um dos problemas da análise estática é em relação a bibliotecas externas, i.e., os retornos das funções de bibliotecas externas não podem ser inferidos. Devido a isso e também às limitações da ferramenta mencionadas anteriormente, diversas recomendações não foram realizadas. Porém, em um contexto geral, de um total de 1.651 instruções dinâmicas, foram recomendadas 743 refatorações (45%). Em um estudo prévio nesses mesmos sistemas [Rodrigues and Terra 2017b], foi possível recomendar 954 refatorações (58%) a partir de uma análise manual conduzida por um especialista. O resultado deste trabalho indica que uma análise estática – mesmo com suas limitações – foi capaz de identificar os comportamentos de 78% dessas instruções dinâmicas sem esforço algum de especialistas.

Discussão: A análise estática desenvolvida neste trabalho apresentou bons resultados quando comparada aos resultados de um trabalho anterior em que foi analisado manualmente cada instrução dinâmica [Rodrigues and Terra 2017b]. No total de 1.651 instruções, a ferramenta `nodyna` recomendou 743 refatorações (22% menor se comparado a análise feita manualmente).

A instrução `send`, a mais utilizada nos projetos analisados, obteve 543 refatorações. Já na análise manual, esse resultado sobe para 680 refatorações (aumento de 25,23%) devido à limitação da ferramenta de inferir retornos de funções declaradas em bibliotecas externas. Em uma análise manual, os retornos de tais funções foram identificados.

As instruções `instance_variable_get` e `instance_variable_set` obtiveram 47 e 107 recomendações, respectivamente. A refatoração é usualmente atrelada à visibilidade da variável, i.e., a instrução dinâmica está sendo utilizada para obter uma variável privada e, portanto, é recomendado a alteração da visibilidade. Na análise manual, foram recomendadas mais cinco refatorações para `instance_variable_get` e 19 para `instance_variable_set` (aumento de 10,63% e 17,76%, respectivamente). Tais refatorações ocorrem devido ao uso de funções que são invocadas ao utilizar herança e, portanto, o algoritmo de inferência de tipos e valores não consegue inferir os parâmetros utilizados.

As instruções `const_get` e `const_set` obtiveram 6 e 39 recomendações, respectivamente. Na análise manual, foram recomendadas mais 26 refatorações para `const_get` e 22 para `const_set` (aumento de 56,41% e 433,34%). O grande aumento de recomendações pela análise manual ocorrem devido a duas situações já citadas anteriormente: (i) inferir retornos de funções declaradas em bibliotecas externas e (ii) uso de funções que são invocadas ao utilizar herança.

Por último, foi possível verificar que as instruções `class_variable_set` e `class_variable_get` não são utilizadas frequentemente. Na única ocorrência da instrução `class_variable_get`, a ferramenta não conseguiu realizar uma recomendação devido a limitação de herança da ferramenta. Entretanto, na análise manual, foi possível rastrear o comportamento de tal instrução. Já na instrução `class_variable_set`, `nodyna` sugeriu uma refatoração, enquanto que a análise manual pôde recomendar duas refatorações. Isso se deve à limitação de herança da ferramenta.

6. Trabalhos relacionados

Furr et al. [Furr et al. 2009b, Furr et al. 2009a, Furr et al. 2009c] desenvolveram diversas ferramentas que contribuíram significativamente para a inferência de tipos, detecção de erros e remoções de instruções dinâmicas em Ruby. A ferramenta DRuby [Furr et al. 2009b] utiliza a estrutura RIL (*Ruby Intermediate Language*) [Furr et al. 2009a] para realizar análises estáticas, pois tal estrutura reduz trechos de código redundantes oferecidas pela linguagem nativa. O algoritmo de inferência de tipos implementado na ferramenta é baseado em restrições, i.e., para a instrução `x.y()`, por exemplo, a variável `x` somente pode ser inferida por classes que possuem o método `y`. A ferramenta PRuby [Furr et al. 2009c], uma extensão de DRuby, combina as abordagens de análise estática e dinâmica para coletar comportamentos de instruções dinâmicas e removê-las. Este estudo diferencia dos estudos de Furr et al. uma vez que a inferência de valores (e de tipos) é totalmente estática.

Em um estudo empírico em projetos JavaScript, foi concluído que a instrução dinâmica `eval` (permite a execução de *strings* como código) geralmente é desnecessária [Richards et al. 2011]. Meawad et al. [Meawad et al. 2012] projetaram uma ferramenta que dinamicamente inspeciona e analisa os parâmetros das chamadas `eval` para sugerir remoções da instrução. Como resultado de uma avaliação, foram removidas mais de 97% das instruções `eval`. Em um outro estudo sobre o mesmo tema, Jensen et al. [Jensen et al. 2012] desenvolveram uma ferramenta que utiliza uma análise de fluxo para remover instruções `eval`. De 44 usos comuns de `eval` coletados em 28 programas, 33 foram convertidos para código estático (75%). Embora `eval` seja uma instrução que oferece grande flexibilidade para o programador, não foi formalizada uma refatoração para essa instrução neste trabalho devida à sua complexidade.

7. Conclusão

Instruções dinâmicas usualmente oferecem ao desenvolvedor maior flexibilidade e generalidade. Em alguns cenários, como no desenvolvimento de *frameworks*, tais instruções permitem a implementação do software ser mais simples. Entretanto, o uso desnecessário das instruções dinâmicas prejudicam o código em relação à compreensibilidade, manutenibilidade, detecção de erros, otimizações, análises estáticas, etc.

Este trabalho, portanto, propôs e formalizou 20 refatorações para a transformação de instruções dinâmicas em estáticas. De forma sucinta, as refatorações podem ser vistas como variações de dois procedimentos comuns: (i) invocação da instrução dinâmica com parâmetros estáticos; e (ii) invocação da instrução dinâmica com parâmetros dinâmicos. Essas refatorações expõem o comportamento de instruções dinâmicas, melhorando a compreensibilidade, manutenibilidade e desempenho. No entanto, conforme pode ser observado, a maioria dessas refatorações aumentam a verbosidade pela adição de código, o que é de fato uma consequência inevitável ao objetivo principal.

Para demonstrar a aplicabilidade das refatorações propostas, um protótipo de um sistema de recomendação, denominado `nodyna`, implementa as refatorações propostas e algoritmos de inferência de tipos e valores para a linguagem Ruby. Em uma avaliação em 28 projetos de código aberto, foi possível recomendar 743 refatorações de instruções dinâmicas (45%). Em um estudo prévio nesses mesmos sistemas [Rodrigues and Terra 2017b], foi possível recomendar 954 refatorações (58%) a par-

tir de uma análise manual conduzida por um especialista. O resultado deste trabalho indica que uma análise estática – mesmo com suas limitações – foi capaz de identificar os comportamentos de 78% dessas instruções dinâmicas sem esforço algum de especialistas.

Como trabalho futuro, propõe-se novas refatorações para as demais instruções dinâmicas (`eval`, `define_method`, `instance_eval`, `etc.`), tratamento das atuais limitações do `nodyna` e uma especificação da semântica por linguagens funcionais [Bonifácio and Borba 2009].

Agradecimentos: Este trabalho foi apoiado pelo CNPq e FAPEMIG.

Referências

[Black 2009] Black, D. A. (2009). *The well-grounded Rubyist*. Manning.

[Bonifácio and Borba 2009] Bonifácio, R. and Borba, P. (2009). Modeling scenario variability as crosscutting mechanisms. In *8th International Conference on Aspect-oriented Software Development (AOSD)*, pages 125–136.

[Callaú et al. 2011] Callaú, O., Robbes, R., Tanter, É., and Röthlisberger, D. (2011). How developers use the dynamic features of programming languages: The case of Smalltalk. In *8th Working Conference on Mining Software Repositories (MSR)*, pages 23–32.

[Callaú et al. 2013] Callaú, O., Robbes, R., Tanter, E., and Röthlisberger, D. (2013). How (and why) developers use the dynamic features of programming languages: The case of Smalltalk. *Empirical Software Engineering*, 18(6):1156–1194.

[Furr et al. 2009a] Furr, M., An, J., Foster, J. S., and Hicks, M. (2009a). The Ruby intermediate language. In *5th Dynamic Languages Symposium (DLS)*, pages 89–98.

[Furr et al. 2009b] Furr, M., An, J., Foster, J. S., and Hicks, M. (2009b). Static type inference for Ruby. In *24th Symposium on Applied Computing (SAC)*, pages 1859–1866.

[Furr et al. 2009c] Furr, M., hoon (David) An, J., and Foster, J. S. (2009c). Profile-guided static typing for dynamic scripting languages. In *24th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 283–300.

[Hills 2015] Hills, M. (2015). Evolution of dynamic feature usage in PHP. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 525–529.

[Holkner and Harland 2009] Holkner, A. and Harland, J. (2009). Evaluating the dynamic behaviour of Python applications. In *32nd Australasian Conference on Computer Science (ACSC)*, pages 19–28.

[Jensen et al. 2012] Jensen, S. H., Jonsson, P. A., and Møller, A. (2012). Remedying the eval that men do. In *21st International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44.

[Meawad et al. 2012] Meawad, F., Richards, G., Morandat, F., and Vitek, J. (2012). Eval begone!: semi-automated removal of eval from JavaScript programs. In *25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 607–620.

[Miranda et al. 2016a] Miranda, S., Rodrigues, E., Valente, M. T., and Terra, R. (2016a). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 15(3):1–34.

[Miranda et al. 2016b] Miranda, S., Valente, M. T., and Terra, R. (2016b). Inferência de tipos em Ruby: Uma comparação entre técnicas de análise estática e dinâmica. In *IV Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, pages 105–112.

[Ren et al. 2013] Ren, B. M., Toman, J., Strickland, T. S., and Foster, J. S. (2013). The Ruby type checker. In *28th Symposium on Applied Computing (SAC)*, pages 1565–1572.

[Richards et al. 2011] Richards, G., Hammer, C., Burg, B., and Vitek, J. (2011). The eval that men do. In *25th European Conference on Object-Oriented Programming (ECOOP)*, pages 52–78.

[Rodrigues and Terra 2017a] Rodrigues, Jr., E. and Terra, R. (2017a). Como desenvolvedores usam instruções dinâmicas? Um estudo em Ruby. In *XXXVI Concurso de Trabalhos de Iniciação Científica (CTIC)*, pages 2492–2501.

[Rodrigues and Terra 2017b] Rodrigues, Jr., E. and Terra, R. (2017b). How do developers use dynamic features? The case of Ruby. *Computer Languages, Systems and Structures*, pages 1–27. (*under minor review*).