Refactorings for Replacing Dynamic Instructions with Static ones: The Case of Ruby

Elder Rodrigues Jr.*
Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP
elder.junior@students.ic.unicamp.br

Rafael Serapilha Durelli
Depto de Ciência da Computação
Universidade Federal de Lavras
Lavras, MG
rafael.durelli@dcc.ufla.br

Raphael Winckler de Bettio
Depto de Ciência da Computação
Universidade Federal de Lavras
Lavras, MG
raphaelwb@dcc.ufla.br

Leonardo Montecchi
Instituto de Computação
Universidade Estadual de Campinas
Campinas, SP
leonardo@ic.unicamp.br

Ricardo Terra
Depto de Ciência da Computação
Universidade Federal de Lavras
Lavras, MG
terra@dcc.ufla.br

ABSTRACT

Dynamic features offered by programming languages provide greater flexibility to the programmer (e.g., dynamic constructions of classes and methods) and reduction of duplicate code snippets. However, the unnecessary use of dynamic features may detract from the code in many ways, such as readability, comprehension, and maintainability of software. Therefore, this paper proposes 20 refactorings that replace dynamic instructions with static ones. In an evaluation on 28 open-source Ruby systems, we could refactor 743 of 1,651 dynamic statements (45%).

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*;

KEYWORDS

Frameworks; refactoring; dynamic statements; dynamic languages

ACM Reference Format:
Elder Rodrigues Jr., Rafael Serapilha Durelli, Raphael Winckler de Bettio, Leonardo Montecchi, and Ricardo Terra. 2018. Refactorings for Replacing Dynamic Instructions with Static ones: The Case of Ruby. In *Proceedings of XXII Simpósio Brasileiro de Linguagens de Programação (SBLP'18).* ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.475/123_4

1 INTRODUÇÃO

Instruções dinâmicas são funções que permitem a invocação, construção e execução de códigos de forma dinâmica, isto é, que dependem de valores externos como variáveis, arquivos ou saídas de microsserviços. A função `send`, por exemplo, permite a execução de um método dinamicamente. Considerando uma invocação de *send(x)* em que *x* pode assumir o valor "foo" ou "bar", então a instrução dinâmica pode invocar o método "foo()" ou "bar()". Tal instrução oferece grande flexibilidade para o desenvolvimento, por exemplo, de *frameworks*, pois quando um método é construído dinamicamente é mais fácil invocá-lo utilizando a função `send`. Nesse contexto, instruções dinâmicas usualmente oferecem ao desenvolvedor maior flexibilidade e generalidade, contudo o uso desnecessário desse tipo de instrução pode prejudicar: (i) abordagens de análises estáticas [3, 4, 9, 11]; (ii) legibilidade, manutenibilidade, complexidade e compreensão do código [19]; (iii) detecção antecipadas de erros, o que esconde erros de tipos [7, 17] ou violações arquiteturais [14]; (iv) otimizações feitas por compiladores [3, 4]; e (v) funcionalidades de IDEs tais como auto-completar e refatorações [19].

Este artigo reivindica que instruções dinâmicas devem ser sim utilizadas, mas especificamente para auxiliar o desenvolvedor em situações em que o código é construído dinamicamente. Por exemplo, em um cenário onde classes e métodos são criados dinamicamente (e.g., criação dinâmica para comunicação com banco de dados), então invocações dos métodos devem ser feitas de forma dinâmica utilizando a função `send`. O foco deste artigo, entretanto, é no uso desnecessário de tais instruções por comodismo ou mesmo por preguiça dos desenvolvedores em escrever código bem estruturado. Um indicativo de uso desnecessário das instruções dinâmicas é quando os valores recebidos por tais instruções podem ser rastreados estaticamente, i.e., sem a necessidade da execução do programa. Nesse contexto, o uso de instruções dinâmicas pode esconder o que de fato está sendo executado, o que pode impactar na compreensibilidade e consequente manutenibilidade do código.

Diante disso, este artigo propõe 20 refatorações para transformar instruções dinâmicas em estáticas. 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 tuplas $[\mathtt{var}, \mathtt{TV} = \{(\mathtt{T}_1, \mathtt{v}_1), ..., (\mathtt{T}_n, \mathtt{v}_n)\}]$ que indica que a variável *var* pode assumir o tipo/valor de cada par ordenado do conjunto `TV`. Analisa-se, então, as instruções dinâmicas a fim de identificar oportunidades de refatoração. Um protótipo, denominado `nodyna`, implementa a recomendação – e não aplicação

*Elder esteve vinculado à Universidade Federal de Lavras durante parte do desenvolvimento deste estudo.

SBLP'18, Setembro 2018, São Carlos, São Paulo Brazil

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

direta – dessas refatorações para a linguagem Ruby. Em uma avaliação em 28 sistemas Ruby de código aberto, nodyna recomendou 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 no intuito de melhorar 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.

As principais contribuições deste artigo são:

- Definir 20 refatorações para transformar instruções dinâmicas em estáticas com o objetivo de diminuir o tempo de manutenção e compreensão do sistema de software; e
- Modelar, implementar, validar e disponibilizar o sistema de recomendação denominado nodyna[1], o qual recomenda as refatorações propostas na linguagem Ruby – embora aplicado para Ruby, é importante salientar que este estudo pode ser facilmente adaptado para outras linguagens dinâmicas, pois o método de inferência de tipos e valores (Seção 3) pode ser reutilizado e, além disso, diversas instruções dinâmicas em Ruby são semelhantes a outras linguagens.

2 INSTRUÇÕES DINÂMICAS EM RUBY

Ruby é uma linguagem 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 [6]. Programas Ruby são desenvolvidos por meio de objetos, uma vez que a linguagem é totalmente orientada a objetos [1]. 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.

```
1  module ModuleA
2    def say_hi
3      puts "Hi"
4    end
5  end
6  class ClassA
7    include ModuleA
8    def say_hello
9      puts "Hello"
10   end
11 end
12 class ClassB < ClassA
13   def say_bye
14     puts "Bye"
15   end
16 end
17 obj = ClassB.new
18 obj.say_hello #Hello
19 obj.say_hi #Hi
20 obj.say_bye #Bye
```

Código 1: Exemplo de código Ruby[2]

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).

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 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.

Baseado em um estudo anterior [20] em que 12 instruções dinâmicas foram analisadas manualmente, este estudo foca nas seguintes sete:[3]

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

O Código 2 ilustra três cenários de usos desnecessário para as instruções dinâmicas. No primeiro cenário, a função `const_set` é utilizada para definir a constante `CONST` (linha 2), porém a instrução dinâmica pode ser facilmente removida bastando declarar a constante normalmente. No segundo cenário, a função `instance_variable_get` é utilizada para obter o valor da variável privada `attr` (linha 14), porém a instrução dinâmica pode ser facilmente removida bastando criar o método `get` para a variável. Por último, a função `send` é utilizada para invocar um método nomeado pela variável `funcName`. No entanto, tal variável é estaticamente rastreável, podendo assumir os valores `qux` ou `bar`. Portanto, o mais indicado é deixar explícito o que é executado no código, utilizando, por exemplo, a estrutura de *switch/when* ao invés de utilizar a função `send`. O uso de instruções dinâmicas com valores estaticamente rastreáveis pode, portanto, indicar uma má estruturação do código (e.g., não utilização de polimorfismo, arquitetura mal planejada, etc.). Por outro lado, podem existir casos em que a instrução dinâmica pode ser utilizada para diminuir a alta verbosidade de um código. Por exemplo, caso o conjunto de possíveis valores da variável `funcName` fosse grande, então a função `send` pode ser utilizada para evitar um código muito verboso.

[1] https://github.com/pqes/nodyna

[2] Um módulo (`module`) em Ruby não possui objetos associados nem classes. Em Ruby, módulos usualmente atuam como *namespace* ou como mecanismo para definir métodos compartilhados entre classes.

[3] É importante destacar que cinco instruções – `define_method` (define um método dinamicamente), `instance_exec` e `instance_eval` (executam um bloco de instruções no contexto de um objeto), `eval` e `class_eval` (avaliam uma *string* como um código Ruby) – não foram consideradas neste estudo e são consideradas trabalhos futuros. Para recomendar a substituição de tais instruções é necessário uma avaliação complexa da estrutura do código. Por exemplo, determinar o que a execução de um bloco de instruções pode alterar na estrutura de classes e métodos ou identificar quais os efeitos da execução dinâmica de um código Ruby expressado por um *string*.

```ruby
class ClassA
  const_set(:CONST, 1)
  def initialize
    @attr = 0
  end
  def foo(funcName)
    send(funcName)
  end
  private
  def qux; return "qux"; end
  def bar; return "bar"; end
end
obj = ClassA.new
obj.instance_variable_get(:@attr)
obj.foo(:qux)
obj.foo(:bar)
```

Código 2: 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 tuplas $[\texttt{var}, \texttt{TV} = \{(T_1, v_1), ..., (T_n, v_n)\}]$ que indica que a variável *var* pode assumir o tipo/valor de cada par ordenado do conjunto TV. Por exemplo, a tupla $[\texttt{m}, \texttt{TV} = \{(\texttt{String}, \text{`}\texttt{foo}\text{`}), (\texttt{String}, \text{`}\texttt{bar}\text{`})\}]$ indica que a variável m pode assumir o tipo *String* com os valores *foo* e *bar*.

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 fazem o uso regular de instruções dinâmicas [20]), o algoritmo de inferência de valores implementado estende um algoritmo de inferência de tipos baseado em análise estática em um estudo anterior [15] (antes apenas os tipos eram mapeados). Basicamente, a partir de atribuições diretas de valores existentes no código fonte, o algoritmo propaga tais atribuições, inferindo valores de outras atribuições. É importante mencionar que o algoritmo implementado é puramente estático, isto é, não é necessário a execução do programa alvo para realizar a inferência, mas herda as limitações de técnicas de análise estática. 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

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:

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

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

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

(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:

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

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

(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.

4 REFATORAÇÕES PARA TRANSFORMAÇÃO DE INSTRUÇÕES DINÂMICAS EM ESTÁTICAS

Com base nas tuplas 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 no intuito de melhorar 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, uma vez que refatorações não podem alterar o comportamento [5, 16]. No entanto, caso o desenvolvedor garanta a completude das refatorações sugeridas, o bloco *else* pode ser removido.

4.1 Descrição das refatorações

Esta seção descreve e e ilustra cada uma das refatorações propostas. A Tabela 1 descreve as funções auxiliares utilizadas para as definições das refatorações, 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 dessas instruções

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

Ref.	*Template*	Pré-condições	Descrição
#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) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then v else instance/class_variable_get(x)
#4	obj.instance_variable_get(x) obj.class_variable_get(x)	isDynamic(x) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case 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) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case 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) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then obj.setV(y) else obj.instance/class_variable_set(x)

Basicamente, uma recomendação 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.

No Código 4, podem ser observadas 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 `@instanceVar1`. Devido ao primeiro parâmetro da instrução ser estático ("`@instanceVar1`") 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 8-15 do Código 5. A condição `else` (linhas 13-15 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 de `instance_variable_set/get`

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

Código 5: Refatorações de `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()`). De forma geral, as refatorações seguem o mesmo princípio do exemplo apresentado anteriormente. Visto isso, os Códigos 6 e 7 ilustram um exemplo real da recomendação de `send` no projeto CocoaPods.

No código 6, pode ser observada a invocação dinâmica de um método nomeado pela variável `file_accessor_key` (linha 5). O algoritmo de inferência de tipos e valores inferiu que tal variável pode assumir os valores `{:source_files, :vendored_frameworks, :vendored_libraries, :resources, resource_bundle_files}`. Portanto, a recomendação é de deixar explícito qual método é invocado e utilizar a instrução *case* (linhas 5–18 do Código 7). Nesse caso, são incrementadas oito linhas de código caso a recomendação seja aplicada. A condição `else` (linhas 16–17) preserva o comportamento do código.

Tabela 3: Refatorações de send

Ref.	*Template*	Pré-condições	Descrição
#1	send(x)	isStatic(x)	x()
#2	obj.send(x)	isStatic(x)	obj.x()
#3	send(x)	isDynamic(x) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then v() else send(x)
#4	obj.send(x)	isDynamic(x) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then obj.v() else obj.send(x)

```
1  def add_file_accessors(file_accessor_key, ...)
2    file_accessors.each do file_accessor
3      pod_name = file_accessor.spec.name
4      local = sandbox.local?(pod_name)
5      paths = file_accessor.send(file_accessor_key)
6      ...
7    end
8  end
```

Código 6: Uso de send

```
1  def add_file_accessors(file_accessor_key, ...)
2    file_accessors.each do file_accessor
3      pod_name = file_accessor.spec.name
4      local = sandbox.local?(pod_name)
5      paths = case file_accessor_key
6        when :source_files
7          file_accessor.source_files()
8        when :vendore_frameworks
9          file_accessor.vendore_frameworks()
10       when :vendored_libraries
11         file_accessor.vendored_libraries()
12       when :resources
13         file_accessor.resources()
14       when :resource_bundle_files
15         file_accessor.resource_bundle_files
16       else
17         file_accessor.send(file_accessor_key)
18       end
19     ...
20   end
21 end
```

Código 7: Recomendação 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`). Novamente, de forma geral, as refatorações seguem o mesmo princípio do exemplo apresentado anteriormente. Visto isso, o Código 8 ilustra um exemplo real quando o algoritmo implementado de inferência de tipos e valores falha em inferir os valores e tipos de uma variável.

No Código 8, pode ser observada a obtenção dinâmica de uma constante nomeada pela variável `subcommand` (linhas 6 e 7). No entanto, o algoritmo de inferência de tipos e valores não foi capaz de inferir os valores e tipos de tal variável. Devido a isso, a função auxiliar `valuesOf` retorna um conjunto vazio e, portanto, não é possível realizar uma recomendação para esse caso.

4.1.4 Observações. Recomendações para instruções dinâmicas que são executadas com variáveis sempre possuem um bloco `else`

```
1  def execute_repl_command(repl_command)
2    if (repl_command != "\n")
3      repl_commands = repl_command.split
4      subcommand = repl_commands.shift.capitalize
5      arguments = repl_commands
6      subcommand_class =
7        Pod::Command::IPC.const_get(subcommand)
8      ...
9    end
10 end
```

Código 8: Uso de `const_set/get`

que executa a instrução dinâmica original. Isso ocorre no intuito de se preservar o comportamento, já que o algoritmo de inferência de tipos não é completo (limitações são discutidas na seção seguinte). No entanto, fica a critério do desenvolvedor remover ou não o bloco `else`.

Por último, 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 para que a recomendação seja válida.

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. Basicamente, `nodyna` analisa instruções dinâmicas para identificar oportunidades de uma das refatorações propostas. A implementação e seu código fonte estão publicamente disponíveis em:

`https://github.com/pqes/nodyna`

Limitações: A implementação atual possui pelo menos quatro limitações: (i) não há uma análise de fluxo de execução; (ii) instruções dinâmicas não são interpretadas; (iii) herança e polimorfismo não são analisados; e (iv) execuções de blocos de instruções não são interpretadas. As limitações citadas podem interferir no algoritmo de inferência de tipos e valores, fazendo com que algumas refatorações não sejam detectadas e, portanto, não sendo recomendadas

Tabela 4: Refatorações de const_set/get

Ref.	*Template*	Pré-condições	Descrição
#1	const_get(x)	isStatic(x)	x
#2	obj.const_get(x)	isStatic(x)	obj::x
#3	const_get(x)	isDynamic(x) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then v else const_get(x)
#4	obj.const_get(x)	isDynamic(x) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case 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) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then v = y else const_set(x, y)
#8	obj.const_set(x, y)	isDynamic(x) ∧ valuesOf(x) ≠ ∅	when x (∀v ∈ valuesOf(x)) case v then obj::v = y else obj.const_set(x, y)

(vide Código 8). De forma simplificada, o Código 9 ilustra todas as limitações citadas e seus possíveis impactos.

```
1  class ClassA
2    def initialize; @attr = "baa"; end
3  end
4  class ClassB < ClassA
5    def qux()
6      func = "ba"
7      eval('@var = "foo"')
8      if(<cond>)
9        func += "a"
10     else
11       func += "r"
12     end
13     send(func)
14     send(@var)
15     send(@attr)
16   end
17   def baa; return "baa"; end
18   def bar; return "bar"; end
19   def foo; return "foo"; end
20 end
21 bloco = Proc.new(){ param
22   obj = ClassB.new().send(param)
23 }
24 bloco.call("qux")
```

Código 9: Limitações da ferramenta implementada

Fluxo de Execução: Qualquer valor em que é necessário uma análise do fluxo de execução não será inferido. Por exemplo, no Código 9, o valor da variável *func* não pode ser inferido pois é necessário uma análise de fluxo de execução para garantir qual o valor final correto. Portanto, a limitação faz com que não seja possível recomendar uma refatoração para o send na linha 13.

Instruções dinâmicas: Execuções de instruções dinâmicas não são consideradas. Por exemplo, no Código 9, a instrução eval insere no escopo a variável @var, mas tal inserção não é visível para a ferramenta implementada. Por isso, não é possível recomendar nenhuma refatoração para o send na linha 14.

Herança e Polimorfismo: Herança e polimorfismo não são considerados. Por exemplo, no Código 9, a classe ClassB herda a variável @attr da classe ClassA. Porém, como herança e polimorfismo ainda não são considerados, então não é possível inferir os valores da variável no contexto da classe ClassB. Portanto, não é feita uma recomendação para o send na linha 15.

Blocos de Instruções: Em Ruby é possível declarar blocos de instruções e executá-los posteriormente. O protótipo implementado não analisa as execuções de tais blocos. Por exemplo, no Código 9, um bloco de instrução é declarado (linhas 21–23) e é executado posteriormente (linha 24). Apesar de ser possível notar que a variável param pode assumir o valor "qux", tal inferência não é feita. Devido a isso, a refatoração para a função send (linha 22) não é recomendada pela ferramenta.

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. Tais sistemas já foram analisados manualmente em trabalhos anteriores [19, 20], permitindo uma avaliação melhor dos resultados da ferramenta. Mais importante, esses sistemas englobam 1.651 instruções dinâmicas, o que é uma representatividade considerável. Assim, executou-se nodyna em cada um dos sistemas cujo resultado é reportado na Tabela 5.

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%). Caso todas as recomendações fossem aplicadas, isso implicaria em um aumento de 2.038 linhas de código, média de 72 linhas por projeto. Em um estudo prévio nesses mesmos sistemas [20], 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.

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**

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 [20]. 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 recomendou 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. [6–8] desenvolveram diversas ferramentas que contribuíram significativamente para a inferência de tipos, detecção de erros e remoções de instruções dinâmicas em Ruby. A ferramenta DRuby [7] utiliza a estrutura RIL (*Ruby Intermediate Language*) [6] para realizar análises estáticas, pois tal estrutura reduz trechos de código redundantes oferecidas pela linguagem nativa. Em particular, este trabalho não utilizou a estrutura RIL, pois não foi encontrada uma documentação completa para percorrer e interpretar

tal estrutura. 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 [8], 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`, que permite a execução de *strings* como código (mesma funcionalidade em Ruby), geralmente é desnecessária [18]. Meawad et al. [13] 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. [12] 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 definida 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 podem prejudicam o código em relação principalmente à compreensibilidade e manutenibilidade.

Este trabalho propôs 20 refatorações para transformar 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 no intuito de melhorar a compreensibilidade e manutenibilidade. 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%). Caso todas as recomendações fossem aplicadas, isso implicaria em um aumento de 2.038 linhas de código, média de 72 linhas por projeto. Em um estudo prévio nesses mesmos sistemas [20], 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.

Como trabalhos futuros, propõe-se a coleta de evidências de que existem benefícios para a aplicação das recomendações de refatorações propostas por meio de uma avaliação com os desenvolvedores. Além disso, devido ao fato de os projetos analisados neste trabalho estarem no *GitHub*, *pull requests* podem ser enviados e entrevistas com os desenvolvedores podem ser realizadas. Também propõe-se formalizar as refatorações sugeridas neste estudo por meio de leis de programação [10] e técnicas formais [2]. Por último, o tratamento das atuais limitações do `nodyna` e novas refatorações para as demais instruções dinâmicas (`eval`, `define_method`, `instance_eval` etc.).

Agradecimentos: Este trabalho foi apoiado pelo CNPq, CAPES e FAPEMIG.

REFERÊNCIAS

[1] David A. Black. 2009. *The well-grounded Rubyist.* Manning.

[2] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. 2004. Algebraic reasoning for object-oriented programming. *Science of Computer Programming* 52, 1-3 (2004), 53–100.

[3] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2011. How Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. In *8th Working Conference on Mining Software Repositories (MSR)*. 23–32.

[4] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. 2013. How (and Why) Developers Use the Dynamic Features of Programming Languages: The Case of Smalltalk. *Empirical Software Engineering* 18, 6 (2013), 1156–1194.

[5] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

[6] Michael Furr, Jong_hoon An, Jeffrey S. Foster, and Michael Hicks. 2009. The Ruby intermediate language. In *5th Dynamic Languages Symposium (DLS)*. 89–98.

[7] Michael Furr, Jong_hoon An, Jeffrey S. Foster, and Michael Hicks. 2009. Static Type Inference for Ruby. In *24th Symposium on Applied Computing (SAC)*. 1859–1866.

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

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

[10] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. 1987. Laws of programming. *Commun. ACM* 30, 8 (1987), 672–686.

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

[12] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the eval that men do. In *21st International Symposium on Software Testing and Analysis (ISSTA)*. 34–44.

[13] Fadi Meawad, Gregor Richards, Floréal Morandat, and Jan Vitek. 2012. Eval begone!: semi-automated removal of eval from JavaScript programs. In *25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 607–620.

[14] Sergio Miranda, Elder Rodrigues, Marco Tulio Valente, and Ricardo Terra. 2016. Architecture Conformance Checking in Dynamically Typed Languages. *Journal of Object Technology* 15, 3 (2016), 1–34.

[15] Sergio Miranda, Marco Tulio Valente, and Ricardo Terra. 2016. 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)*. 105–112.

[16] William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[17] Brianna M. Ren, John Toman, T. Stephen Strickland, and Jeffrey S. Foster. 2013. The Ruby type checker. In *28th Symposium on Applied Computing (SAC)*. 1565–1572.

[18] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The eval that men do. In *25th European Conference on Object-Oriented Programming (ECOOP)*. 52–78.

[19] Elder Rodrigues, Jr. and Ricardo Terra. 2017. Como Desenvolvedores Usam Instruções Dinâmicas? Um Estudo em Ruby. In *XXXVI Concurso de Trabalhos de Iniciação Científica (CTIC)*. 2492–2501.

[20] Elder Rodrigues, Jr. and Ricardo Terra. 2018. How Do Developers Use Dynamic Features? The Case of Ruby. *Computer Languages, Systems and Structures* 53 (2018), 1–25.