Como Desenvolvedores Usam Instruções Dinâmicas? Um Estudo em Ruby

Elder Rodrigues Jr, Ricardo Terra

Departamento de Ciência da Computação,
Universidade Federal de Lavras (UFLA), Brasil

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

Resumo. *As instruções dinâmicas, um dos recursos fornecidos pelas linguagens dinâmicas, permitem, em tempo de execução, executar strings como expressões, definir e invocar métodos dinamicamente, etc. Contudo, o uso exacerbado de tais instruções podem impactar negativamente no desempenho do programa, prejudicar a precisão de técnicas de análises estáticas no código e dificultar otimizações feitas pelo compilador. Neste trabalho, portanto, é investigado como os desenvolvedores usam as instruções dinâmicas baseado em dez projetos Ruby de código aberto. Os principais resultados foram: (i) em média, instruções dinâmicas correspondem a 0,53% do total de instruções e metade das instruções dinâmicas utilizadas correspondem ao uso de* `send`*; (ii) foi identificado que 897 das 1.856 instruções dinâmicas (48,3%) analisadas são viáveis de serem convertidas para códigos estáticos; (iii) foram identificados, classificados e ilustrados 23 cenários onde os desenvolvedores optam por instruções dinâmicas ao invés de estáticas; e (iv) instruções dinâmicas permitem que o desenvolvedor escreva, em poucas linhas, códigos que serão responsáveis por adaptar o software para diversos cenários.*

1. Introdução

Linguagens dinâmicas fornecem recursos que aceleram a codificação para tornar o processo de desenvolvimento mais produtivo [5]. As instruções dinâmicas, fornecidas por tais linguagens, permitem aos desenvolvedores, em tempo de execução, avaliar *strings* como expressões, definir e invocar métodos dinamicamente, etc.

O uso exacerbado das instruções dinâmicas pode ser indesejável para um projeto, pois prejudicam: (i) a precisão de técnicas de análises estáticas [1, 6, 7, 8]; (ii) legibilidade, facilidade de entendimento, manutenibilidade, etc. [12]; (iii) detecção antecipada de erros e otimizações feitas pelos compiladores [3]; (iv) precisão de algoritmos para inferência de tipos, o que pode esconder erros de tipos [13] ou erros arquiteturais [11]; e (v) recursos de uma IDE, tais como auto-completar e refatorações [12].

Esta iniciação científica, portanto, investiga como desenvolvedores utilizam instruções dinâmicas em 10 projetos Ruby de código aberto (o que totaliza mais de 500KLOC). O estudo empírico aborda as seguintes quatro questões de pesquisa:

1. *Com que frequência desenvolvedores utilizam instruções dinâmicas?* Em média, instruções dinâmicas correspondem a 0,53% do total de instruções em um projeto Ruby e praticamente 50% delas são instruções `send`.

2. *Qual a complexidade em se remover as instruções dinâmicas?* Foi identificado que 897 das 1.856 instruções dinâmicas (48,3%) podem ser facilmente convertidas para instruções estáticas. Essa porcentagem varia de 28,2% para o domínio de *Plug-ins para Rails* (o que é esperado devido a flexibilidade que *plug-ins* devem ter) a 62,6% e 64,2% para *Aplicações Web* e *Gerenciadores*, respectivamente.
3. *Quando desenvolvedores utilizam instruções dinâmicas?* Foram identificados, classificados e ilustrados 23 cenários onde os desenvolvedores optam pela utilização de instruções dinâmicas ao invés de estáticas, e.g., foi descoberto que desenvolvedores utilizam instruções dinâmicas para acessar métodos e atributos privados (20,4%), o que pode evidenciar falhas no projeto arquitetural.
4. *Por que desenvolvedores utilizam instruções dinâmicas?* De acordo com as análises feitas nas 1.856 instruções dinâmicas, foram encontradas as vantagens de um desenvolvimento mais rápido e com menos linhas de código. De fato, as instruções dinâmicas permitem com que o desenvolvedor escreva, em poucas linhas, códigos que serão responsáveis por adaptar o software para diversos cenários.

O restante deste documento está organizado como a seguir. Seção 2 discute o desenvolvimento do estudo empírico (questões de pesquisa, sistemas utilizados, metodologia, ferramentas, ameaças à validade, etc.). Seção 3 apresenta os trabalhos relacionados e a Seção 4 conclui o trabalho e lista as publicações resultantes desta iniciação científica.

2. Estudo empírico

2.1. Questões de pesquisa

Este estudo foi projetado para abordar as seguintes questões de pesquisa:

RQ #1 - *Com que frequência desenvolvedores utilizam instruções dinâmicas?*

RQ #2 - *Qual a complexidade em se remover as instruções dinâmicas?*

RQ #3 - *Quando desenvolvedores utilizam instruções dinâmicas?*

RQ #4 - *Por que desenvolvedores utilizam instruções dinâmicas?*

2.2. Instruções dinâmicas em Ruby

Cada uma das instruções dinâmicas analisadas neste artigo permitem definir e obter constantes dinamicamente (`const_set` e `const_get`, respectivamente), definir métodos (`define_method`), executar *string* como código, executar blocos com ou sem parâmetros (`instance_eval` e `instance_exec`, respectivamente) e invocar métodos (`send`). Outras linguagens também fornecem essas funcionalidades, embora com diferentes funções.

2.3. Sistemas analisados

A Tabela 1 apresenta os 10 sistemas considerados neste estudo, incluindo seus nomes, versões, linhas de código (LOC), número de arquivos *rb* e número de bibliotecas. O critério para a seleção dos 10 projetos foi os que possuíam o mais alto número de instruções dinâmicas dos 30 projetos Ruby com mais estrelas.[1] No total, foram analisados mais de 500 KLOC em 8.342 arquivos *rb*.

[1]https://github.com/search?l=&o=desc&q=language:Ruby+stars:>1&ref=advsearch&s=stars&type=-Repositories, pesquisado em agosto de 2015.

Tabela 1. Sistemas Analisados

Projeto e versão	LOC	# de arquivos rb	# de Gems
Active Admin (v1.0.0.pre1)	6.054	154	42
diaspora* (v0.5.2.0)	15.269	376	128
Discourse (vlatestes-realease)	36.951	657	101
GitLab (v7.14.1)	29.345	574	137
Homebrew (8278b89)	133.322	3.429	4
Paperclip (v4.3.0)	3.081	59	34
Rails (v4.2.4)	55.530	849	82
RailsAdmin (v0.7.0)	4.624	111	48
Ruby (v2_2_3)	169.300	1.076	0
Spree (v3.0.4)	103.596	1.057	6

2.4. Suporte ferramental

Devido ao fato de a análise feita neste estudo ser manual, foi desenvolvida uma ferramenta – chamada `nodyna` – que auxilia na investigação das instruções dinâmicas de um modo mais organizado e efetivo.[2] A ferramenta realiza uma análise textual em todo o código fonte do projeto e possui as seguintes funcionalidades: (i) *setup*: a cada instrução dinâmica encontrada no projeto, é feita uma marcação padrão em forma de comentário no próprio código indicando que a instrução ainda não foi classificada; (ii) *show_locations*: lista os arquivos que possui a instrução pesquisada e que já foi marcada; (iii) *show_locations_without_classification*: lista os arquivos que possui a instrução pesquisada e que já foi marcada, mas ainda não foi classificada; e (iv) *show_classifications*: lista todas as classificações feitas e a quantidade de instruções que cada classificação possui.

2.5. RQ #1 - *Com que frequência desenvolvedores utilizam instruções dinâmicas?*

Nessa questão de pesquisa, o objetivo é de medir a frequência de uso das instruções dinâmicas em Ruby.

Metodologia: Comparou-se o número de características dinâmicas com o número total de recursos da linguagem nos sistemas analisados. Em tal contexto, recursos da linguagem são expressões, instruções e declarações (atribuições, chamadas de métodos, laços, etc.). Já características dinâmicas são invocações ou construções dinâmicas. Especificamente em Ruby, Como descrito na Seção 2.2, este estudo considera as seguintes funções: (i) `const_get`, (ii) `const_set`, (iii) `define_method`, (iv) `eval`, (v) `instance_eval`, (vi) `instance_exec` e (vii) `send`. Um exemplo é ilustrado na Listagem 1.

```
1 class Test
2   def bar
3     x = Z.new
4     if x.send(:foo)
5       y = 3
6     end
7   end
8 end
```

Listagem 1. Número de instruções dinâmicas

Nesse exemplo, existem sete tipos de instruções: (i) definição de classe na linha 1, (ii) definição de método na linha 2, (iii) instanciação do tipo `Z` na linha 3, (iv) atribuição

[2]A ferramenta, seu código fonte e os sistemas analisados estão publicamente disponíveis em: https://github.com/rterrabh/nodyna

para a variável `x` na linha 3, (v) condicional na linha 4, (vi) invocação dinâmica do método `foo` também na linha 4, e (vii) atribuição para a variável `y` na linha 5. Portanto, a relação entre o número de instruções dinâmicas com o uções nesse código é de 1/7 = 14,3%.

Resultados: Conforme reportado na Tabela 2, em média instruções dinâmicas correspondem a 0,53% do total de instruções em projeto Ruby. Contudo, existem sistemas com um número de instruções dinâmicas relativamente alto (em torno de 1%) , tais como Active Admin, Paperclip e RailsAdmin. Paperclip, por exemplo, é um aplicativo para gerenciar *uploads* utilizando a biblioteca ActiveRecord, o que necessita de ser bem flexível. Baseado neste estudo, quanto mais um sistema deva ser adaptável, mais instruções dinâmicas ele possuirá.

A instrução `send` é responsável por quase 50% das instruções dinâmicas. Tal método permite chamar dinamicamente qualquer função recebendo como parâmetro uma variável que contém o nome da função a ser chamada. Por exemplo, é possível escrever a = 2.send('*', 3) ao invés de a = 2 * 3. Isso facilita para os programadores chamarem funções que são construídas em tempo de execução.

Projeto	# inst.	# inst. dinâmicas	eval	instance_exec	instance_eval	define_method	const_set	const_get	send
Active Admin	8.978	112 (1,25%)	2 (1,79%)	33 (29,46%)	1 (0,89%)	11 (9,82%)	1 (0,89%)	2 (1,79%)	62 (55,36%)
diaspora*	34.931	88 (0,25%)	1 (1,14%)	1 (1,14%)	29 (32,95%)	2 (2,27%)	0 (0,00%)	1 (1,14%)	54 (61,36%)
Discourse	64.897	216 (0,33%)	25 (11,57%)	2 (0,93%)	2 (0,93%)	37 (17,13%)	0 (0,00%)	2 (0,93%)	148 (68,51%)
GitLab	52.102	59 (0,11%)	1 (1,69%)	0 (0,00%)	0 (0,00%)	4 (6,78%)	1 (1,69%)	3 (5,09%)	50 (84,75%)
Homebrew	169.676	78 (0,05%)	0 (0,00%)	0 (0,00%)	15 (19,23%)	16 (20,51%)	2 (2,56%)	3 (3,85%)	42 (53,85%)
Paperclip	4.714	67 (1,42%)	1 (1,49%)	0 (0,00%)	2 (2.99%)	5 (7,46%)	0 (0,00%)	5 (7,46%)	54 (80,60%)
Rails	84.408	428 (0,51%)	8 (1,87%)	22 (5,14%)	16 (3,74%)	71 (16,59%)	11 (2,57%)	24 (5,61%)	276 (64,48%)
RailsAdmin	7.998	76 (0,95%)	0 (0,00%)	0 (0,00%)	21 (27,63%)	4 (5,26%)	0 (0,00%)	1 (1,32%)	50 (65,79%)
Ruby	248.522	544 (0,22%)	93 (17,10%)	29 (5,33%)	166 (30,51%)	49 (9,01%)	63 (11,58%)	39 (7,17%)	105 (19,30%)
Spree	103.596	188 (0,18%)	1 (0,53%)	8 (4,26%)	6 (3,19%)	17 (9,04%)	0 (0,00%)	1 (0,53%)	155 (82.45%)
Total	**779.822**	**1.856**	**132**	**95**	**258**	**216**	**78**	**81**	**996**

2.6. RQ #2 - *Qual a complexidade em se remover as instruções dinâmicas?*

A complexidade de remoção da instrução dinâmica é um fator crucial para este estudo, uma vez que a necessidade do uso da instrução dinâmica está ligada diretamente a sua complexidade de remoção. Quanto mais fácil é remover uma instrução dinâmica, menos necessária ela é.

Metodologia: Com o auxílio do `nodyna`, cada uma das 1.856 instruções dinâmicas foi classificada nos seguintes níveis de complexidade de remoção: *trivial* (mudanças locais e sem novas instruções), *baixo* (mudanças locais na classe e sem novas instruções), *médio* (mudanças locais na classe com adição de novas instruções) e *alto* (procedimentos complexos e que requerem mudanças em muitas classes ou em arquivos externos).

Resultados: A Tabela 3 apresenta os resultados. Em um contexto geral, o número de instruções que são viáveis[3] para remoção é de 48%. Em uma análise por domínio, os sistemas analisados foram agrupados da seguinte forma: *Aplicação Web* (Discourse, Diaspora e Spree), *Gerenciadores* (GitLab e Homebrew) e *Plug-ins para Rails*.

Foram criados grupos isolados para o projetos *Ruby* e para o projeto *Rails*, pois esses projetos possuem um grande número de instruções dinâmicas e podem influenciar

[3]Foi adotada a terminologia "*viáveis*" para aquelas instruções dinâmicas que sua complexidade de remoção é *trivial*, *baixa* ou *média*.

Tabela 3. Análise das complexidades de remoção das instruções dinâmicas

		Active Admin	Diaspora*	Discourse	GitLab	Homebrew	Paperclip	Rails	RailsAdmin	Ruby	Spree	Total
`const_get`	Trivial	0	0	0	0	1	0	2	0	4	0	**7** (0,38%)
	Baixa	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Média	0	1	1	0	1	0	2	0	7	1	**13** (0,70%)
	Alta	2	0	1	3	1	5	20	1	28	0	**61** (3,29%)
`const_set`	Trivial	0	0	0	0	1	0	3	0	45	0	**49** (2,64%)
	Baixa	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Média	0	0	0	0	0	0	2	0	8	0	**10** (0,54%)
	Alta	1	0	0	1	1	0	6	0	10	0	**19** (1,02%)
`define_method`	Trivial	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Baixa	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Média	4	2	32	3	14	1	37	3	37	6	**139** (7,49%)
	Alta	7	0	5	1	2	4	34	1	12	11	**77** (4,15%)
`eval`	Trivial	0	0	0	1	0	0	0	0	0	0	**1** (0,05%)
	Baixa	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Média	0	1	1	0	0	0	0	0	3	0	**5** (0,27%)
	Alta	2	0	24	0	0	1	8	0	90	1	**126** (6,79%)
`instance_eval`	Trivial	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Baixa	0	24	0	0	0	0	0	0	8	1	**33** (1,78%)
	Média	0	0	0	0	4	0	3	1	95	0	**103** (5,55%)
	Alta	1	5	2	0	11	2	13	20	63	5	**122** (6,57%)
`instance_exec`	Trivial	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Baixa	0	0	0	0	0	0	0	0	0	0	**0** (0,00%)
	Média	0	1	0	0	0	0	0	0	0	0	**1** (0,05%)
	Alta	33	0	2	0	0	0	22	0	29	8	**94** (5,07%)
`send`	Trivial	9	3	5	5	1	17	22	3	9	15	**89** (4,80%)
	Baixa	4	25	9	6	0	7	29	1	22	44	**147** (7,92%)
	Média	10	20	67	23	28	6	56	6	35	49	**300** (16,16%)
	Alta	39	6	67	16	13	24	169	40	39	47	**460** (24,78%)

bastante nos resultados dos demais. A porcentagem de complexidade `alta` para remoção das instruções dinâmicas em *Ruby* e *Rails* são de 49,8% e 63,6% respectivamente. Devido à flexibilidade que *plug-ins* devam ter, o domínio *Plug-ins para Rails* apresentou alta complexidade de remoção para 71,8% de suas instruções dinâmicas. Os domínios *Aplicações Web* e *Gerenciadores*, contudo, apresentam uma baixa taxa de instruções dinâmicas complexas de serem removidas (37,4% e 35,8%, respectivamente).

2.7. RQ #3 - *Quando desenvolvedores utilizam instruções dinâmicas?*

Essa questão de pesquisa tem como objetivo identificar, classificar e ilustrar os cenários onde os desenvolvedores utilizam instruções dinâmicas ao invés de estáticas.

Metodologia: Com o auxílio da ferramenta `nodyna`, investigou-se as 1.856 instruções dinâmicas dos sistemas analisados no intuito de identificar por que desenvolvedores optam por instruções dinâmicas.

Resultados: Foram identificados 23 cenários conforme pode ser observado na Tabela 4. Devido a restrições de espaço, este documento apresenta a sumarização dos resultados (Tabela 4) e uma discussão dos cenários relacionados a cada uma das instruções dinâmicas. No entanto, uma descrição detalhada e um exemplo de cada cenário pode ser encontrada em: `http://www.dcc.ufla.br/~terra/papers/2017_ctic`

- `const_get`: É mais utilizada para que métodos obtenham constantes de um modo mais flexível, uma vez que mais de 50% do uso de tal função está relacionada ao cenário CG-2 (*variáveis dinâmicas*). O cenário CG-1 (*valores estáticos*), que representa o mau uso da função, é responsável por 8,64% do uso de `const_get`.
- `const_set`: É mais utilizada para criar constantes a partir de valores estáticos. Isso indica que os desenvolvedores utilizam `const_set` de forma desnecessária. Ao invés de utilizarem tal função, basta declarar a constante normalmente.

Tabela 4. Cenários de uso de instruções dinâmicas

Instrução dinâmica	Cenário	Complexidade				Total
		Trivial	Baixo	Médio	Alto	
`const_set`	CG-1 (valores estáticos)	47	0	0	0	**47** (2,53%)
	CG-2 (valores dinâmicos)	0	0	3	19	**22** (1,19%)
	CG-3 (lista)	2	0	7	0	**9** (0,48%)
`const_get`	CS-1 (valores estáticos)	7	0	0	0	**7** (0.38%)
	CS-2 (valores dinâmicos)	0	0	5	37	**42** (2.26%)
	CS-3 (lista)	0	0	8	24	**32** (1,73%)
`define_method`	DM-1 (lista)	0	0	77	6	**83** (4,47%)
	DM-2 (eventos)	0	0	62	71	**133** (7,17%)
`eval`	EV-1 (definição de método)	1	0	2	14	**17** (0.92%)
	EV-2 (valores dinâmicos)	0	0	0	70	**70** (3.77%)
	EV-3 (métodos privados)	0	0	0	11	**11** (0.59%)
	EV-4 (escopo)	0	0	0	21	**21** (1.13%)
	EV-5 (definição de classe)	0	0	2	3	**5** (0.27%)
	EV-6 (definição de variável)	0	0	1	7	**8** (0,43%)
`instance_eval`	IEV-1 (acesso privado)	0	33	64	41	**138** (7,44%)
	IEV-2 (definição de método)	0	0	38	7	**45** (2,42%)
	IEV-3 (execução de bloco)	0	0	1	74	**75** (4,04%)
`instance_exec`	IEX-1 (bloco sem parâmetros)	0	0	1	36	**37** (1.99%)
	IEX-2 (bloco com parâmetros)	0	0	0	58	**58** (3,13%)
`send`	SD-1 (métodos públicos)	88	0	0	0	**88** (4,74%)
	SD-2 (lista)	1	2	116	67	**186** (10,02%)
	SD-3 (variáveis dinâmicas)	0	9	141	342	**492** (26,51%)
	SD-4 (métodos privados)	0	136	43	51	**230** (12,39%)

- `define_method`: É muito utilizada para criação de métodos durante a execução do programa (invocação de métodos, invocação de método inexistente, etc.). Por outro lado, a criação de diversos métodos com sintaxe semelhante possui, na maioria dos casos, uma conversão fácil para código estático.
- `eval`: É comumente utilizada para fornecer mais flexibilidade, devido ao fato de que mais de 50% do uso está relacionado ao cenário *EV-2 (valores dinâmicos)*. Tal flexibilidade faz com que a conversão para código estático seja mais complexa (como observado nos resultados da maioria dos cenários em que praticamente todos possuem uma alta porcentagem de classificações com complexidade *alta*). Além disso, `eval` também é utilizado para deixar o desenvolvimento do software mais flexível, permitindo criação de classes, métodos e variáveis (22,73%) em tempo de execução. Finalmente, foi demonstrado que os desenvolvedores também utilizam `eval` para acessar métodos privados (8,33%), o que pode indicar violações na arquitetura planejada.
- `instance_eval`: É comumente utilizado para acessar variáveis e métodos privados. Como já mencionado, tal acesso pode indicar violações na arquitetura planejada, já que variáveis e métodos privados são acessados fora do contexto da classe em que pertencem.
- `instance_exec`: Desenvolvedores a utilizam para fornecer aos métodos mais flexibilidade, o que faz com que a conversão para códigos estáticos tenha alta complexidade.
- `send`: Foi possível identificar que 8,84% dessas instruções são utilizadas para chamar métodos públicos, o que indica o uso desnecessário da função. Também foi identificado que 23,09% do uso de `send` é para chamar métodos privados, o que pode indicar violações na arquitetura planejada.

2.8. RQ #4 - *Por que desenvolvedores utilizam instruções dinâmicas?*

A análise de vantagens e desvantagens das instruções dinâmicas permite que seja possível identificar o porquê dos desenvolvedores utilizarem tais instruções.

Metodologia: Com o suporte da ferramenta `nodyna`, buscou-se por todas as instruções analisadas neste estudo. Basicamente, para cada uma das 1.856 instruções dinâmicas, foram analisadas as potenciais razões que motivam os desenvolvedores a utilizarem as instruções dinâmicas.

Resultados: Foi possível identificar três principais razões para o uso das mesmas: (i) *flexibilidade*: instruções dinâmicas fornecem aos desenvolvedores a possibilidade de escreverem, em poucas linhas, códigos que são responsáveis em adaptar o código para diversos tipos de cenários. Por exemplo, identificou-se cenários em que as instruções dinâmicas foram utilizadas para permitir que o software opere em qualquer modelo de banco de dados; (ii) *tamanho:* a conversão de instruções dinâmicas para estática pode aumentar o tamanho do código em dezenas de linhas. Portanto, uma outra vantagem para o uso de instruções dinâmicas é o desenvolvimento do software de uma forma mais rápida; e (iii) *violações arquiteturais*: os desenvolvedores utilizam, em diversos momentos, as instruções dinâmicas para obterem acesso a variáveis e métodos privados utilizando `eval`, `instance_eval`, `instance_exec` e `send`. Isso pode indicar violações no projeto arquitetural e impactar na manutenibilidade, escalabilidade, portabilidade, etc. [11].

2.9. Discussão

Existem três observações complementares para se discutir. Primeira, foi identificado que em alguns cenários os desenvolvedores utilizam instruções dinâmicas para seguir um padrão estabelecido em *gems* (bibliotecas em Ruby) antigas. Nesses casos, a substituição para instruções estáticas é trivial. Segundo, desenvolvedores usualmente dependem das instruções dinâmicas para obterem acesso a variáveis e métodos privados. Isso pode indicar más decisões no projeto arquitetural, uma vez que se métodos e atributos estão sendo acessados fora do contexto de uma classe, então eles deveriam ser públicos. Terceiro, desenvolvedores usualmente definem métodos que recebem, através dos parâmetros, nomes de outros métodos que possivelmente serão chamados. Nesses casos, o uso das instruções dinâmicas (tal como `send`) deixa o desenvolvimento do software mais rápido, embora a complexidade de remoção dessas instruções oscile entre complexidade *média* e *alta*.

2.10. Ameaças à validade

É possível identificar duas ameaças à validade para este estudo. Em primeiro lugar, este estudo investiga o uso de instruções dinâmicas em 10 projetos Ruby. Como usual em estudos empíricos na área de engenharia de software, não é possível afirmar que o resultado apresentado neste trabalho será o mesmo para outros sistemas. Contudo, há heterogeneidade entre os projetos, uma vez que eles foram desenvolvidos por diferentes equipes. Em segundo lugar, as classificações das instruções dinâmicas foram feitas por um único desenvolvedor Ruby experiente (e também o primeiro autor deste estudo). Para mitigar tal ameaça, todos os arquivos e projetos utilizados neste estudo estão publicamente disponíveis para fins de conferência.[2]

3. Trabalhos Relacionados

Os trabalhos relacionados estão divididos em três grupos: (i) *Instruções dinâmicas*; (ii) *Eval em JavaScript*; e (iii) *Ferramentas em Ruby*.

Instruções dinâmicas: Hills et al. [7] investigaram quais instruções dinâmicas são usadas, o quão frequente elas são e como elas estão distribuídas ao longo dos arquivos em

sistemas PHP. Como resultado, eles estaticamente inferiram que 61% das chamadas poderiam ser resolvidas por códigos estáticos enquanto que neste estudo foi possível inferir 48%. Hills [6] também investigou a evolução das instruções dinâmicas em projetos PHP. O autor classificou as instruções dinâmicas em três categorias: (i) *variable features*, (ii) *magic methods* e (iii) `eval`. Conectando com este estudo, *variable features* e *magic methods* são similares ao uso de `send` em Ruby e o `eval` do PHP apresenta os mesmos comportamentos ao do Ruby. Ele concluiu que `eval` está sendo menos usado ao longo do tempo, mas *variable features* e *magic methods* estão sendo mais utilizados. Tal resultado é semelhante ao apresentado neste estudo, uma vez que desenvolvedores de Ruby usam `send` muito mais que `eval` (996 vs. 132). Callaú et al. [1] conduziram um estudo empírico em um grande projeto que usa a linguagem Smalltalk. Similar a este estudo (mas em outra linguagem), seus objetivos eram de verificar quais são as principais razões do uso de instruções dinâmicas e se tais instruções podem ser convertidas para códigos estáticos. Eles concluíram que instruções dinâmicas não são muito utilizadas (o que colabora com os resultados deste trabalho) e algumas instruções dinâmicas são mais utilizadas em alguns tipos de aplicações (o que não foi possível concluir neste estudo). Também foi reportado que 40,7% das instruções dinâmicas são estaticamente rastreáveis, enquanto que neste estudo o porcentual foi de 48,3%.

Eval em JavaScript: Richards et al. [15] conduziram um estudo sobre o comportamento dinâmico em JavaScript. Baseado na execução de programas que monitoram as atividades de um sistema, foi descoberto que a instrução `eval` é comumente utilizado. Em um estudo mais focado em `eval`, Richards et al. [14] desenvolveram uma ferramenta que automaticamente carrega mais de 10.000 *sites* para obter informações sobre operações executadas pelo `eval`. Os autores afirmam que, em diversos cenários, o uso de `eval` é desnecessário e pode ser facilmente substituído por instruções estáticas, o que diferencia deste estudo, uma vez que 95% das instruções `eval` são complexas de serem removidas. Meawad et al. [10] projetaram uma ferramenta que dinamicamente inspeciona e analisa os parâmetros das chamadas `eval` para sugerir remoções da instrução. A ferramenta baseia-se em um algoritmo de inferência gramatical para converter o `eval` em códigos estáticos. Foi possível remover mais de 97% de instruções `eval`. Em um outro estudo sobre o mesmo tema, Jensen et al. [9] desenvolveram uma ferramenta que utiliza uma análise de fluxo para remover instruções `eval`. De 44 instruções `eval` coletadas em 28 programas, 33 foram convertidas para códigos estáticos (75%). Em comparação a este trabalho, apenas foi possível converter 4,55% de instruções `eval` (o que evidência a importância de uma análise dinâmica para algumas instruções dinâmicas).

Ferramentas em Ruby: Furr et al. [2] propuseram a ferramenta *Ruby Intermediate Language* (RIL), que deixa o código Ruby mais fácil de ser analisado por ferramentas e desenvolvedores. Por exemplo, o RIL consegue obter os parâmetros do `eval` através de uma análise dinâmica e substituir o `eval` por todos os possíveis casos de execução. Dessa forma, a execução do `eval` se tornaria explícita e, portanto, facilitando o entendimento para os desenvolvedores e deixando a tarefa de manutenção do código mais fácil. Furr et al. [4] também desenvolveram PRuby, uma ferramenta que baseia-se nos perfis de utilização das instruções dinâmicas para removê-las. Eles relataram que os desenvolvedores utilizam mais o `eval` do que o `send` (27 *vs.* 15). No entanto, em comparação a este estudo, o `send` é muito mais comum do que o `eval` (132 *vs.* 996).

4. Conclusão

Esta iniciação científica investigou o uso de instruções dinâmicas em 10 projetos Ruby de código aberto. Em média, tais instruções correspondem a 0,53% nos sistemas analisados. Entretanto, de acordo com os estudos deste trabalho, quanto mais um sistema deve ser genérico, maior vai ser o uso de instruções dinâmicas (e.g., 1,42% no sistema Paperclip). Além disso, em média, 48% das instruções analisadas neste trabalho podem ser convertidas para códigos estáticos. Em um estudo por domínio, tal resultado pode aumentar para mais de 60% ao lidar com os projetos nos domínios de *Aplicação Web* e *Gerenciadores*.

Foi verificado que os desenvolvedores frequentemente utilizam instruções dinâmicas para acessarem métodos e atributos privados (20,4%), o que pode indicar violações na arquitetura planejada. Foram também identificadas três razões para o uso de instruções dinâmicas: *flexibilidade* (permitem com que o desenvolvedores escrevam, em poucas linhas, códigos que são responsáveis em adaptar o programa em diversos cenários, *tamanho* (o código é usualmente menor) e *violações arquiteturais* (permitem o acesso a variáveis e métodos privados).

Como já mencionado, a instrução `send` foi a mais utilizada. Essa instrução permite que a invocação de métodos definidos dinamicamente seja mais prática e ágil. Esse resultado não é restrito apenas para a linguagem Ruby. No trabalho de Callaú et al. [1], por exemplo, também foi demonstrado que instruções classificadas como *message sending* (equivalente ao `send` em Ruby) também são as mais utilizadas. Além disso, a funcionalidade da função `send` pode ser encontrada na API *Java Reflection*, o que significa que tal funcionalidade não está limitada apenas às linguagens dinâmicas.

Como trabalhos futuros, planeja-se descobrir associações usando análises estáticas (por exemplo, algumas instruções dinâmicas são mais usadas em certos tipos de aplicações?). Além disso, também planeja-se estender a ferramenta `nodyna` para analisar estaticamente um projeto Ruby de forma a realizar inferências de tipos. Com tal funcionalidade, a ferramenta poderá ser capaz de recomendar ao usuário conversões das instruções dinâmicas para estáticas através de heurísticas formalizadas para cada instrução dinâmica estudada neste artigo.

Finalmente, este estudo contempla a análise de 1.856 instruções dinâmicas nos 10 projetos Ruby com mais estrelas no GitHub. A análise manual de tais instruções não é trivial, sendo necessário um estudo prévio sobre cada projeto para entender o contexto do uso de cada instrução dinâmica. A principal dificuldade para realizar este estudo foi devido ao fato de que em diversos momentos instruções dinâmicas deixam a análise e estudo do código mais complexa e, portanto, exigem mais tempo e experiência.

Publicações resultantes: Durante a iniciação científica, o aluno participou efetivamente dos seguintes artigos:

- Elder Rodrigues Jr, Ricardo Terra. How Do Developers Use Dynamic Features? A Case Study on Ruby. In *11th International Symposium on Empirical Software Engineering and Measurement (EMSE)*, pp. 1-10, 2017 (***submetido, Qualis A2***).
  Esse estudo reflete exatamente o projeto de IC conduzido pelo aluno, o qual foi descrito neste documento.
- Sergio Miranda, Elder Rodrigues Jr, Marco Tulio Valente, Ricardo Terra. Architecture Conformance Checking in Dynamically Typed Languages. *Journal of Ob-*

ject Technology, 15(3):1-34, 2016. (***publicado, Qualis B2***).

A contribuição do aluno foi dupla: (i) o aluno conduziu um estudo empírico de contagem de instruções dinâmicas e (ii) o aluno atuou – de forma autônoma – no projeto e implementação de uma visão arquitetural.

Agradecimentos: Este trabalho foi apoiado pelo CNPq.

Referências

[1] Callaú, O., Robbes, R., Tanter, É., and Röthlisberger, D. (2013). How (and why) developers use the dynamic features of programming languages: the case of Smalltalk. In *8th Working Conference on Mining Software Repositories (MSR)*, pages 1156–1194.

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

[3] 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.

[4] 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.

[5] Hanenberg, S. (2010). An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *25th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 22–35.

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

[7] Hills, M., Klint, P., and Vinju, J. (2013). An empirical study of PHP feature usage: a static analysis perspective. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 325–335.

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

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

[10] 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.

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

[12] Palsberg, J. and Schwartzbach, M. I. (1991). Object-oriented type inference. In *6th International conference on object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 146–161.

[13] 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.

[14] 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.

[15] Richards, G., Lebresne, S., Burg, B., and Vitek, J. (2010). An analysis of the dynamic behavior of JavaScript programs. In *31st Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12.