Inferência de Tipos em Ruby: Uma comparação entre técnicas de análise estática e dinâmica

Sergio Miranda[1], **Marco Tulio Valente**[1], **Ricardo Terra**[2]

[1]Universidade Federal de Minas Gerais, Belo Horizonte, Brasil

[2]Universidade Federal de Lavras, Lavras, Brasil

{sergio.miranda,mtov}@dcc.ufmg.br, terra@dcc.ufla.br

Resumo. *Informações sobre tipos que variáveis podem assumir é importante para desenvolvedores de software, uma vez que os apoiam no entendimento do código, aumentam a confiança para realizar atividades de refatoração, auxiliam na detecção de erros, etc. No entanto, como característica intrínseca de linguagens dinamicamente tipadas, não há definições explícitas de tipos de variáveis. Quando necessário, a inferência de tipos é realizado por técnicas de análise dinâmica, as quais requerem a execução do sistema e possuem alto custo computacional. Diante disso, este artigo avalia o uso de análise estática—em comparação com análise dinâmica—na inferência de tipos em linguagens dinamicamente tipadas. Uma análise em seis sistemas de código aberto demonstrou que, em média, um algoritmo elementar por análise estática já é capaz de inferir 44% dos tipos. Uma análise qualitativa apontou sete melhorias (duas simples, duas moderadas e três complexas) que devem ser aplicadas ao algoritmo elementar para se obter resultados equivalentes aos obtidos por análise dinâmica.*

1. Introdução

Linguagens dinâmicas não impõem restrições de tipos durante o desenvolvimento, ou seja, execuções corretas do sistema de software não são impedidas de forma prematura. Apesar de agilizar o desenvolvimento, essa característica traz certas desvantagens [Agesen et al. 1995]. Informações de tipo podem apoiar positivamente diversas tarefas de manutenção, evolução e compreensão de software, pois (a) tornam o código mais legível e auto-documentável; (b) facilitam a implementação de ferramentas de análise de código; (c) tornam refatorações mais seguras e confiáveis; (d) permitem a construção de melhores IDEs (por exemplo, com suporte a recursos de auto-completar); e (e) permitem a detecção antecipada de erros, garantindo que todas mensagens enviadas sejam entendidas [Palsberg and Schwartzbach 1991].

Informações sobre os tipos das variáveis não são fáceis de serem obtidas em linguagens dinâmicas em comparação com linguagens estaticamente tipadas. Quando necessário, a inferência de tipos ocorre por meio de técnicas de análise dinâmica, as quais requerem a execução do sistema e possuem alto custo computacional [Agesen and Holzle 1995]. Isso dificulta o trabalho de ferramentas que auxiliam desenvolvedores em atividades de melhoria da qualidade do sistema de software. Por exemplo, ferramentas para garantia de conformidade arquitetural, ferramentas para realizar refatoração automaticamente, etc., podem se beneficiar de informações sobre tipos. Especificamente, ArchRuby – um verificador de conformidade arquitetural para a linguagem Ruby – utiliza uma heurística *estática* de inferência de tipos [Miranda et al. 2016, Miranda et al. 2015].

Neste artigo, a *questão de pesquisa* se resume em verificar os desafios do uso de análise estática para inferência de tipos em linguagens dinamicamente tipadas. Portanto, este artigo compara as informações de tipos coletadas através de análise estática com informações coletadas através de análise dinâmica. Para análise estática, utilizou-se um algoritmo para inferência de tipos com uma heurística simples [Miranda et al. 2016]. Já para análise dinâmica, foram utilizados dados coletados durante a execução do sistema de software. O objetivo é comparar as duas situações e prover melhorias para que a análise estática possa gerar resultados equivalentes àqueles proporcionados por técnicas de análise dinâmica. Uma análise quantitativa em seis sistemas de código aberto demonstrou que, em média, um algoritmo elementar por análise estática é capaz de inferir 44% dos tipos. Complementarmente, uma análise qualitativa apontou sete melhorias que devem ser aplicadas para se obter resultados equivalentes aos obtidos por análise dinâmica.

O restante deste artigo está organizado como a seguir. A Seção 2 detalha a heurística de inferência de tipos utilizada. A Seção 3 apresenta detalhes da avaliação conduzida, tais como questões de pesquisa propostas, projeto do estudo e resultados encontrados. A Seção 4 apresenta trabalhos relacionados. Por fim, a Seção 5 conclui.

2. Inferência de Tipos

Existem diversos algoritmos para realizar inferência de tipos em linguagens dinâmicas [Palsberg and Schwartzbach 1991, Agesen and Holzle 1995], sendo que alguns necessitam de anotações para auxiliar no processo [Furr et al. 2009]. Consequentemente, a necessidade de anotar o código dificulta o entendimento e a adoção desses algoritmos. Este artigo, portanto, parte de um algoritmo proposto para a linguagem Ruby que é baseado em análise estática [Miranda et al. 2016, Miranda et al. 2015]. Embora dinamicamente tipada, é possível inferir em Ruby parte dos tipos de variáveis e parâmetros formais. O algoritmo escolhido utiliza-se de uma heurística – mais especificamente, uma simplificação da heurística formalizada por Furr et al. [Furr et al. 2009] – que visa construir um conjunto `TYPES` de triplas $[\texttt{method}, \texttt{var\_name}, \texttt{type}]$, onde `type` é um dos possíveis tipos inferidos para a variável ou parâmetro formal `var_name` do método `method`. Esse conjunto é construído de acordo com a seguinte definição recursiva:

i) **Base**: Para cada inferência direta (e.g., instanciação) de um tipo `T` de uma variável `x` em um método `f`, então $[\texttt{f}, \texttt{x}, \texttt{T}] \in \texttt{TYPES}$.

ii) **Passo recursivo**: Se $[\texttt{f}, \texttt{x}, \texttt{T}] \in \texttt{TYPES}$ e existir em `f` uma chamada $\texttt{g}(\texttt{x})$, então $[\texttt{g}, \texttt{y}, \texttt{T}] \in \texttt{TYPES}$, onde `y` é o nome do parâmetro formal de `g`. Esse passo é aplicado até um ponto fixo ser atingido, i.e., até nenhum novo elemento ser adicionado ao conjunto de `TYPES`.

A Listagem 1 ilustra o funcionamento da heurística proposta [Miranda et al. 2016]. Ao executar o passo base do algoritmo, `TYPES` é inicializado com $[\texttt{A::f}, \texttt{x}, \texttt{Foo}]$, $[\texttt{A::f}, \texttt{b}, \texttt{B}]$, $[\texttt{A::f}, \texttt{self}, \texttt{A}]$, $[\texttt{B::g}, \texttt{c}, \texttt{C}]$ e $[\texttt{C::h}, \texttt{d}, \texttt{D}]$ já que são as inferências diretas. Na primeira aplicação do passo recursivo, as triplas $[\texttt{B::g}, \texttt{x}, \texttt{Foo}]$ e $[\texttt{B::g}, \texttt{z}, \texttt{A}]$ são incluídas em `TYPES`, uma vez que se conhece o tipo das variáveis `x` e `self` na chamada de `g`. Na segunda aplicação do passo recursivo, as triplas $[\texttt{C::h}, \texttt{y}, \texttt{Foo}]$ e $[\texttt{C::h}, \texttt{y}, \texttt{A}]$ são incluídas em `TYPES`, uma vez que se conhece o tipo das variáveis `x` e `z` na chamada de `h`. Na terceira aplicação do passo recursivo, as triplas $[\texttt{D::m}, \texttt{k}, \texttt{Foo}]$ e $[\texttt{D::m}, \texttt{k}, \texttt{A}]$ são incluídas em `TYPES` (onde `k` é o nome do parâmetro em `D`), uma vez que se

conhece o tipo da variável `y` na chamada de `m`. A aplicação do passo recursivo se repete até que nenhuma nova tripla seja adicionada ao conjunto.

```
1  class A                    class B                    class C
2    def f                      def g(x z)                 def h(y)
3      x = Foo.new                c = C.new                d = D.new
4      b = B.new                  c.h(x)                   d.m(y)
5      b.g(x,self)                c.h(z)                  end
6    end                        end                      end
7  end                        end
```

Listagem 1. Código para ilustração da heurística de inferência de tipo

Nesse exemplo, é importante notar que o parâmetro formal `y` do método `C::h` pode ser do tipo `A` ou `Foo`. Isso indica que o mecanismo de propagação de tipos deve considerar todos os potenciais tipos de uma variável ou parâmetro formal.

3. Estudo Comparativo

Nesta seção é avaliado o algoritmo descrito na seção anterior utilizando seis sistemas de código aberto desenvolvidos em Ruby. A avaliação compara informações de tipos geradas através da execução dos testes automatizados dos sistemas (i.e., análise dinâmica) com as informações de tipos geradas pelo algoritmo descrito na Seção 2 (i.e., análise estática).

3.1. Questões de Pesquisa

Este estudo tem como objetivo responder às seguintes questões de pesquisa:

- ***QP #1*** – Qual a acurácia de abordagens estáticas de inferência de tipos em comparação com as dinâmicas?
- ***QP #2*** – Como é possível ampliar a acurácia de abordagens estáticas de inferência de tipo?

3.2. Projeto do estudo

Dataset: Avaliou-se o algoritmo de inferência de tipos em seis sistemas de código aberto desenvolvidos em Ruby. Os critérios de escolha foram: (i) possuir testes automatizados e (ii) ser executável com a versão 2.1 ou superior da linguagem Ruby, pois a máquina virtual da linguagem só oferece maneiras de inspecionar o *runtime* a partir de tal versão. A Tabela 1 apresenta os sistemas escolhidos juntamente com informações sobre a porcentagem de cobertura de testes[1] e quantidade de linhas de código.

Oráculo: Considerou-se como oráculo os tipos inferidos pelos testes automatizados dos sistemas. Embora essa decisão possa gerar falsos negativos, procurou-se minimizar tal problema selecionando sistemas com elevada porcentagem de cobertura de testes (média de 89%, conforme reportado na Tabela 1).

Métricas: As informações de tipos foram coletadas de dois modos: (i) pela execução dos testes automatizados dos sistemas (i.e., inspecionando o *runtime* e armazenando todos os tipos gerados durante a execução) e (ii) pela execução do algoritmo estático descrito na Seção 2. Foram gerados pares ordenados para representar os tipos coletados. Suponha que

[1]Foi utilizada a ferramenta SimpleCov, disponível em: https://github.com/colszowka/simplecov.

Tabela 1. Sistemas avaliados

Sistema	Cobertura de Testes	LOC
Capistrano	94%	1.779
CarrierWave	81%	2.636
Devise	97%	3.683
Resque	84%	2.315
Sass	95%	13.609
Vagrant	87%	8.429
Média	**89%**	**5.408**

uma variável `x` é do tipo `A`, logo a representação é dada pelo par ordenado $(\texttt{x}, \{\texttt{A}\})$. Dessa maneira, são propostas duas métricas para auxiliar na comparação dos resultados. A primeira, denominada $Recall_1$ e formalizada na Equação 1, tem como objetivo mensurar a quantidade de tipos que o algoritmo analisado conseguiu inferir de forma exata quando comparado com os tipos obtidos durante a execução dos testes automatizados.

$$Recall_1 = \frac{\mid Static \cap Dynamic \mid}{\mid Dynamic \mid} \tag{1}$$

Para ilustrar $\texttt{Recall}_1$, assuma as seguintes tuplas coletadas durante a execução dos testes automatizados, $(\texttt{a}, \{\texttt{X}\})$, $(\texttt{b}, \{\texttt{Y}\})$ e $(\texttt{c}, \{\texttt{W}, \texttt{Z}\})$. Em seguida, assuma que as tuplas $(\texttt{a}, \{\texttt{X}\})$, $(\texttt{b}, \{\texttt{Y}\})$ e $(\texttt{c}, \{\texttt{W}\})$ foram obtidas pelo algoritmo estático em avaliação. Consequentemente, ao realizar o cálculo de $\texttt{Recall}_1$, o valor encontrado é 67% (2/3), pois apenas as tuplas $(\texttt{a}, \{\texttt{X}\})$ e $(\texttt{b}, \{\texttt{Y}\})$ estão presentes na interseção entre as duas abordagens.

De forma complementar, também é calculado uma segunda métrica denominada $Recall_2$ e formalizada na Equação 2. Essa métrica tem como objetivo analisar a quantidade de variáveis que o algoritmo estático conseguiu coletar, mesmo que parcialmente. A função $variáveis$ retorna o conjunto de variáveis que tiveram tipos obtidos.

$$Recall_2 = \frac{\mid variáveis(Static) \cap variáveis(Dynamic) \mid}{\mid Dynamic \mid} \tag{2}$$

Reconsidere o exemplo anterior para ilustrar o cálculo do valor de $\texttt{Recall}_1$. É importante notar que o algoritmo avaliado inferiu para a variável `c` apenas o tipo $\{\texttt{W}\}$. Dessa maneira, o cálculo de $\texttt{Recall}_2$ seria 100% (3/3), já que o algoritmo avaliado conseguiu inferir pelo menos um tipo para todas as três variáveis (`a`, `b` e `c`).

3.3. Resultados

QP #1 – Qual a acurácia de abordagens estáticas de inferência de tipos em comparação com as dinâmicas?

Após a execução dos testes automatizados e do algoritmo estático em avaliação foram calculados os valores para as métricas descritas na Seção 3.2. A Tabela 2 apresenta os resultados encontrados para os seis sistemas avaliados. Na média, o valor para $\texttt{Recall}_1$ é de 44,4%, já para $\texttt{Recall}_2$ esse valor sobe para 58,1%. Os números entre parênteses representam os valores para o numerador e denominador das fórmulas de $\texttt{Recall}_1$ e $\texttt{Recall}_2$. O maior $\texttt{Recall}_1$, 50,8%, foi para o sistema `CarrierWave`, em que 62 das 122 variáveis foram inferidas pela abordagem estática. Já para $\texttt{Recall}_2$, o sistema `Sass` obteve o maior resultado (58,4%), em que 135 variáveis foram inferidas pela abordagem estática pelo menos parcialmente.

Tabela 2. Resultados

Sistema	$Recall_1$	$Recall_2$
Capistrano	39,0% (30/77)	45,5% (35/77)
CarrierWave	50,8% (62/122)	57,4% (70/122)
Devise	43,8% (155/354)	56,5% (200/354)
Resque	45,0% (9/20)	55,0% (11/20)
Sass	46,0% (500/1.088)	58,4% (635/1.088)
Vagrant	40,8% (220/539)	55,7% (300/539)
Média	**44,4%**	**58,1%**

A Listagem 2 ilustra uma situação onde o algoritmo analisado não foi capaz de inferir o tipo de uma variável para o sistema `Vagrant`.

```
1  def filter(command)
2    command_filters.each do |c|
3      command = c.filter(command) if c.accept?(command)
4    end
5    command
6  end
7  def create_command_filters
8    [].tap do |filters|
9      @@cmd_filters.each do |cmd|
10       require_relative "command_filters/#{cmd}"
11       class_name = "VagrantPlugins::CommunicatorWinRM::
12                     CommandFilters::#{cmd.capitalize}"
13       filters << Module.const_get(class_name).new
14     end
15   end
16 end
```

Listagem 2. Código de exemplo do sistema Vagrant

O *loop* da linha 2 percorre o arranjo populado no método `create_command_filters` (linha 7) para atribuir um valor à variável `command` (linha 3). Ainda, o método `create_command_filters` utiliza técnicas de reflexão para atribuir valores ao arranjo (linhas 11-13), ou seja, os valores só serão conhecidos em tempo de execução. Logo, pode ser complexo ou ate mésmo inexequível para uma análise estática inferir o tipo que será atribuído a tal variável.

De forma similar, a Listagem 3 ilustra um segundo exemplo onde não foi possível detectar o tipo de uma variável para o sistema Capistrano.

```
1  def fetch(key, default=nil, &block)
2    value = fetch_for(key, default, &block)
3    while callable_without_parameters?(value)
4      value = set(key, value.call)
5    end
6    return value
7  end
```

Listagem 3. Código de exemplo do sistema Capistrano

A variável `value` é inicializada na linha 2. Com o valor retornado pelo método `fetch_for`. No entanto, a variável `value` pode ter seu valor alterado no *loop* execu-

tado entre as linhas 3-5. Similarmente à situação apresentada no exemplo anterior, a variável `value` também tem seu valor definido de forma dinâmica, impondo dificuldade para detecção através de análise estática de código.

Por último, a Listagem 4 demonstra uma situação no sistema Devise, onde o valor de uma variável é obtido através de acesso a estrutura de dados `Hash`. A linha 5 atribui, à variável `skip`, o valor que está armazenado na chave `:skip_helpers` do *hash* `options`. Uma análise estática deveria armazenar os valores que são passados como chave e valor para tal estrutura para ser capaz de detectar qual o tipo atribuído à variável `skip`.

```
1  def default_used_helpers(options)
2    singularizer = lambda { |s| s.to_s.singularize.to_sym }
3    if options[:skip_helpers] == true
4      @used_helpers = @used_routes
5    elsif skip = options[:skip_helpers]
6      @used_helpers = self.routes - Array(skip).map(&singularizer)
7    else
8      @used_helpers = self.routes
9    end
10 end
```

Listagem 4. Código de exemplo do sistema Devise

QP #2 – Como é possível ampliar a acurácia de abordagens estáticas de inferência de tipo?

Uma análise qualitativa dos resultados indicam possíveis mudanças que podem auxiliar na melhora dos resultados obtidos. Essas mudanças são classificadas de acordo com a sua complexidade:

- *Simples*: Considerar a funcionalidade de `include` e `extend` oferecida pela linguagem Ruby, as quais implementam o suporte a *mixins* para implementar herança múltipla [Bracha and Cook 1990]. Assim, algoritmos de análise estática deveriam simular o funcionamento dessas funcionalidades armazenando todos os métodos definidos em módulos que são incluídos em classes através da utilização de `include` e `extend`.
- *Moderada*: Considerar a execução e retorno de blocos, os quais são considerados funções de primeira ordem em Ruby, podendo ser armazenados em variáveis e passados como parâmetros. Assim, o processo de análise estática deveria acompanhar as mudanças que são realizadas pela execução de blocos.
- *Complexa*: Considerar valores armazenados em estruturas de dados, tais como `Array`, `Hash`, `Set`, etc. Como Ruby inclui reflexão e avaliação dinâmica de código (e.g., eval), o algoritmo deveria ser capaz de analisar instruções dinâmicas. Por exemplo, `instance_eval`, `class_eval` e `define_method` são funcionalidades que alteram o programa em tempo de execução. Deve-se também considerar a atribuição de valores a variáveis de instância de objetos, o que requer que a análise estática tenha ciência do contexto de execução e do fluxo de atribuições.

A Tabela 3 reporta os resultados de uma análise manual do impacto das mudanças propostas nos sistemas avaliados.

Tabela 3. Análise de impacto das mudanças propostas no valor de `Recall`$_1$

	$Recall_1$								
	Trivial	**Simples**		**Moderada**		**Complexa**			
Sistema	**base**	**include**	**extend**	**retorno blocos**	**execução blocos**	**valores em estrutura**	**instr. dinâmica**	**fluxo de execução em variáveis de instância**	**Total**
Capistrano	39,0%	5,2%	1,3%	14,3%	13,0%	15,6%	1,3%	10,3%	**100%**
CarrierWave	50,8%	6,6%	1,6%	11,5%	13,1%	9,8%	2,5%	4,1%	**100%**
Devise	43,8%	2,8%	1,4%	12,7%	15,5%	9,0%	6,3%	8,5%	**100%**
Resque	45,0%	0%	0%	20,0%	15,0%	10,0%	0%	10,0%	**100%**
Sass	46,0%	0,6%	0,6%	13,8%	18,4%	12,9%	3,6%	4,1%	**100%**
Vagrant	40,8%	2,8%	0,4%	13,0%	22,3%	11,1%	2,2%	7,4%	**100%**
Média	**44,4%**	**2,0%**	**0,7%**	**13,4%**	**18,4%**	**11,7%**	**3,5%**	**5,9%**	**100%**

Note que com a implementação das modificações consideradas simples e moderadas é possível elevar o valor médio de `Recall`$_1$ para 78,9% (44,4+2,0+0,7+13,4+18,4), um aumento considerável para a análise estática. Ainda, adicionando as modificações consideradas complexas esse valor seria elevado para 100% (78,9+11,7+3,5+5,9). Como resultado mais significativo, é importante mencionar que, se implantadas as melhorias de complexidade moderada (retorno e execução de blocos), o `Recall`$_1$ elevaria de 44,4% para 76,2%. Logo, seriam essas as melhorias que trariam o maior benefício imediato a algoritmos estáticos de inferência de tipo.

4. Trabalhos Relacionados

Outros trabalhos investigaram o problema de se aplicar análise estática para realizar inferência de tipos em linguagens dinâmicas. Em um dos primeiros trabalhos, Palsberg e Schwartzbach [Palsberg and Schwartzbach 1991] investigam o uso de análise estática de tipos em uma linguagem dinâmica inspirada em Smalltalk. Por ser uma linguagem fictícia, construída para ilustrar os conceitos apresentados no trabalho, várias funcionalidades de linguagens dinâmicas são desconsideradas (e.g., reflexão, funções de primeira ordem, etc.). Neste trabalho, por outro lado, utilizou-se uma linguagem real sem desconsiderar quaisquer de suas funcionalidades.

Em um estudo recente com Ruby, Furr et al. avaliam o uso de algoritmo de inferência de tipos de forma estática, juntamente com o auxílio de anotações, para auxiliar desenvolvedores a capturarem erros sem a necessidade de executar o programa [Furr et al. 2009]. Neste artigo, o algoritmo estático avaliado é uma versão simples e não invasiva (sem uso de anotações) do algoritmo proposto pelos autores.

Morison [Morrison 2006] desenvolveu um algoritmo de inferência de tipos para Ruby que foi integrado ao IDE RadRails, utilizada por programadores que utilizam o *framework* Rails. Em linhas gerais, o algoritmo realiza análise de fluxo para sugerir os possíveis métodos que podem ser invocados por um determinado objeto. Tal análise é similar à ideia proposta pelo algoritmo apresentado neste artigo, porém se diferencia por analisar fluxos de atribuições que são possíveis de serem feitos a variáveis de instância.

5. Conclusão

Linguagens dinâmicas não impõem restrições de tipos, porém a informação dos tipos que um objeto pode assumir auxilia os mantenedores de software em diversas tarefas, tais como refatorações, correções de *bugs* e detecção de violações arquiteturais. Diante disso, neste trabalho foram comparadas abordagens estática e dinâmica na inferência de tipos em seis sistemas de código aberto obtendo os seguintes resultados: (i) o algoritmo de análise estática foi capaz de inferir, na média, 44,4% dos tipos obtidos pela análise dinâmica e (ii) foram também sugeridas sete melhorias que, se aplicadas ao algoritmo de análise estática, obteria resultados equivalentes aos obtidos por análise dinâmica.

Como trabalho futuro, planeja-se executar os testes em um conjunto maior de sistemas para se obter uma maior confiabilidade estatística na extrapolação dos resultados. Posteriormente, pretende-se implementar as melhorias simples e moderadas, o que elevaria a acurácia apresentada pelo algoritmo de análise estática para 78,9%. Ainda, avaliar o desempenho do algoritmo estático e conduzir o mesmo estudo para outras linguagens dinâmicas – tais como Python e JavaScript – , cujas comunidades podem se beneficiar com informações de tipos.

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

Referências

[Agesen and Holzle 1995] Agesen, O. and Holzle, U. (1995). Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *10th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 91–107.

[Agesen et al. 1995] Agesen, O., Palsberg, J., and Schwartzbach, M. I. (1995). Type inference of self: Analysis of objects with dynamic and multiple inheritance. *Software: Practice and Experience*, 25(9):975–995.

[Bracha and Cook 1990] Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *5th Conference on Object-Oriented Programming: System Languages, and Applications (OOPSLA)*, pages 303–311.

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

[Miranda et al. 2016] 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.

[Miranda et al. 2015] Miranda, S., Valente, M. T., and Terra, R. (2015). Conformidade e visualização arquitetural em linguagens dinâmicas. In *18th Ibero-American Conference on Software Engineering (CIbSE), Software Engineering Technologies (SET) Track*, pages 137–150.

[Morrison 2006] Morrison, J. (2006). Type inference in Ruby. In *Google Summer of Code Project*.

[Palsberg and Schwartzbach 1991] Palsberg, J. and Schwartzbach, M. I. (1991). Object-oriented type inference. In *6th Conference on Object-Oriented Programming: System Languages, and Applications (OOPSLA)*, pages 146–161.