Uso de técnicas adaptativas para manutenção da consistência de coleções baseada em igualdade

SOFIATO, B.*; ROCHA, R. L. A.*
*LTA – Laboratório de Linguagem e Técnicas Adaptativas - Escola Politécnica da USP
E-mail: bruno.sofiato@gmail.com, rlarocha@usp.br

***Resumo*— Linguagens modernas disponibilizam ao desenvolvedor um amplo leque de coleções e algoritmos altamente otimizados. Apesar de sua importância, algumas dessas bibliotecas de coleções apresentam uma falha importante: coleções baseadas em igualdade e coleções convencionais são intercambiáveis, o que potencialmente acarreta falhas, uma vez que a primeira não permite mutações de seus elementos. Devido ao seu grau de utilização, mudanças na hierarquia dessas coleções são inviáveis, uma vez que introduziriam incompatibilidades cuja correção seria custosa. Este artigo tem como objetivo o estudo do impacto da adoção de coleções reindexáveis (i.e. capazes de lidar com mutações em seus elementos) no desempenho geral de programas, bem como a influência da adoção de técnicas adaptativas na mitigação destes impactos. Para a medição do impacto no desempenho do uso dessas coleções, um experimento baseada na suíte de testes de desempenho DaCapo foi conduzido. Resultados obtidos apontam para uma queda expressiva de desempenho que é mitigada quando técnicas adaptativas são empregadas.**

I. Introdução

Um dos pilares da programação orientada a objetos é o conceito de polimorfismo, cuja etimologia remete ao grego *polimorphos* (muitas formas, em tradução livre). Através desse conceito é possível a substituição de uma implementação de um objeto por outra distinta, sem quaisquer alterações nos elementos que o utilizam.

Dentre as diversas formas de polimorfismo, encontra-se o chamado Polimorfismo de Subtipagem [1], com o qual é estabelecida uma relação entre tipo e subtipo. Uma das formas de se formalizar essa relação é através da noção de subtipo introduzida por Liskov e Wing [2] e seu princípio de substituição.

O Princípio da Substituição de Liskov estabelece algumas características que devem ser observadas. Falhas na sua manutenção podem acarretar mudanças inesperadas de comportamento quando existe uma substituição de instâncias de um tipo e por instâncias de um subtipo.

A biblioteca de coleções da linguagem Java [3] define um tipo básico (*Collection*), do qual todas as coleções são subtipos. Esse tipo base define algumas operações que podem ser invocadas em qualquer coleção. Entre essas operações, encontram-se as operações de inclusão de elementos. Apesar de parecer sólida à primeira vista, essa modelagem tem um problema: em razão de características de suas implementações, coleções baseadas em igualdade[1] (como por exemplo conjuntos) podem ter como elementos somente elementos imutáveis. Logo, a definição de um tipo de coleção básico englobando todas as coleções configura uma violação do Princípio da Substituição de Liskov.

Uma solução possível consiste na refatoração da biblioteca de coleções de modo a que coleções baseadas de igualdade tenham tipos distintos onde é permitida apenas a inserção de objetos imutáveis em coleções baseadas em igualdade. O acesso à elementos de coleções poderia ser realizado através de um tipo comum a todas coleções (por exemplo, através de Iteradores [4]) cujas operações se restringiriam a operações de acesso.

Essa solução se mostra impraticável, uma vez que acarreta na coexistência de duas bibliotecas distintas de coleções potencialmente incompatíveis. Outro desafio é a inexistência de suporte a qualificadores de tipos customizados[2] na linguagem Java. Informalmente, qualificadores de tipos permitem uma categorização adicional de tipos. Um exemplo de qualificador de tipos é a palavra-chave *const* [8], existente na linguagem C++: variáveis (ou parâmetros) qualificadas como *const* só podem ter seu valor atribuído por valores qualificado da mesma forma.

Uma outra solução é a utilização de coleções reindexáveis, onde os algoritmos convencionais são acrescido por algoritmos adicionais que exibam um comportamento alternativo quando elementos não são encontrados pelos meios tradicionais. Essa abordagem porém traz alguns desafios: os algoritmos auxiliares fazem uma varredura linear em todos os elementos a procura de elementos que não foram encontrado pelos algoritmos principais. Em outras palavras, a complexidade temporal [9] das operações de busca e remoção degradam, no melhor caso, de $\mathcal{O}(1)$ para $\mathcal{O}(n)$.

Uma forma de mitigação dos impactos na complexidade temporal consiste na adoção de técnicas adaptativas. De forma geral, o uso dessas técnicas consiste em se instanciar coleções

[1] Vale notar que a problemática aqui apresentada é também aplicável a coleções ordenadas. De modo geral, coleções ordenadas mantêm a ordem no momento da inserção de um elemento. Logo, mudanças subsequentes à inserção podem deixar a coleção em estado inconsistente.

[2] A linguagem Java incorpora, a partir de sua versão 1.8, o conceito de anotações de tipo. Esse recurso permite a implementação de sistemas de tipos plugáveis [5], que por sua vez podem dar suporte a qualificadores de tipos customizados. Um exemplo de sistema de tipos que utiliza essa abordagem é o *Checker Framework* [6], [7] (disponível em http://types.cs.washington.edu/checker-framework/).

reindexáveis apenas quando seus elemento sejam passíveis de mutação.

Tendo-se em vista esse cenário, tem-se como principal objetivo deste trabalho o estudo do impacto da adoção de coleções reindexáveis em programas de computador, bem como os ganhos relacionados ao uso de técnicas adaptativas no que tange a mitigação desses impactos. Para isso, é realizado um estudo empírico no qual o impacto da adoção de coleções reindexáveis (com e sem o uso de técnicas adaptativas) nos programas incluídos na suíte de testes de desempenho DaCapo [10] (versão 9.12-bach) é medido.

Segundo Nelson et al. [11], cerca de 40% dos elementos inseridos em coleções baseadas em igualdade são passíveis de mutação. Além disso, apenas 4% dos elementos sofrem mutações que acarretam inconsistências nas coleções onde estão incluídos. Intuitivamente, pode-se esperar um ganho sensível de desempenho quando técnicas adaptativas são utilizadas de modo a coleções reindexáveis só serem instanciadas quando necessário. Apesar da determinação da mutabilidade de um objeto (ou tipo) ser um problema indecidível (i.e. não pode ser resolvido através de algoritmos), é possível a criação de um procedimento que decida se um objeto é imutável. Nesse caso, coleções tradicionais (i.e. sem os algoritmos alternativos) são empregadas.

Este artigo é estruturado da seguinte maneira: uma introdução, descrições básicas sobre os conceitos fundamentais de igualdade, mutação e seus impactos em coleções (respectivamente, Seções II, III e IV). Na Seção V são descritas em detalhes as técnicas de reindexação utilizadas, sendo também apresentada uma análise da complexidade temporal e espacial de ambas as técnicas. A Seção expõe os detalhes de implementação do protótipo utilizado no experimento (Seção), cujo resultados são apresentados e discutidos na Seção VIII. Por fim, são discutidos trabalhos correlatos e futuros (Seções IX e X) e uma breve conclusão (Seção XI).

II. Igualdade

O conceito de igualdade é um conceito usado diariamente quando se deseja comparar e determinar a equivalência entre elementos. Segundo Tarski [12], a semântica da igualdade e comparação depende do domínio no qual esses elementos operam.

Para ilustrar o conceito de igual pode-se imaginar uma classe de objetos chamada $Coordenada$, cujas instâncias representam o conceito geográfico de coordenada. Instâncias dessa classe são compostas de dois atributos: a latitude e a longitude – posições relativa ao Equador (orientação norte-sul) e ao meridiano de *Greenwich* (orientação leste-oeste), respectivamente. Duas coordenadas são consideradas equivalentes quando têm os mesmos valores de latitude e longitude. Nesse caso, pode-se dizer que o estado de igualdade (ou equivalência) de uma coordenada é composto pela sua latitude e longitude.

Uma pequena digressão sobre a definição de estado de igualdade em questão se faz necessária. Nota-se o viés algébrico da definição (i.e. o estado é composto de partes que por sua vez também são compostas por partes menores). Pode-se porém se efetuar uma redução de um elemento (ou uma expressão) arbitrário à uma forma algébrica, conforme demonstrado por Gödel [13]. Apesar de uma procedimento geral ser incomputável, é possível a implementação de funções desse tipo para classes específicas de objetos. Por exemplo, pode-se descrever o estado de equivalência de um algoritmo de ordenação através de uma 1-upla cujo único elemento representa a estabilidade[3] do algoritmo em questão.

A. Coleções Baseadas em Igualdade

Coleções são estruturas de dados amplamente utilizadas em programas de computador, independentemente do paradigma empregado. Seu principal papel é o agrupamento de elementos relacionados. Pode-se citar como exemplos de coleções as listas ligadas, os vetores, e as tabelas de espalhamento [9].

Pode-se identificar entre as coleções um grupo de coleções que têm uma característica em comum: o uso do estado de igualdade de seus elementos. Essa categoria de coleções é denominada de coleções baseadas na igualdade e tem como membros coleções como conjuntos (que, de maneira análoga aos construtos matemáticos de mesmo nome, não permitem elementos duplicados) e as Tabelas de Espalhamento (também conhecidas como Tabelas *Hash* ou Dicionários).

Tabelas de espalhamento foram inventadas por Hans Peter Luhn em meados de 1953 enquanto trabalhava na IBM [14]. Elas têm um funcionamento simples e engenhoso. Uma tabela de espalhamento é composta por vetores auxiliares. No momento da inserção de um novo elemento, é decidido, através de um procedimento determinístico, em qual dos vetores internos o elemento será incluído. No momento de uma busca, esse mesmo procedimento é utilizado para a obtenção do vetor interno onde a busca será efetivamente realizada. Logo, a busca é restrita a apenas um subconjunto de todos os elementos incluídos na tabela.

O procedimento de escolha do vetor interno depende de uma função denominada função de espalhamento (ou função *hash*). Em termos informais, existe uma relação entre o estado de igualdade de um objeto e seu código *hash*: que objetos equivalentes deve ter o mesmo código *hash*. Caso valores diferentes fossem obtidos para objetos semelhantes, buscas subsequentes seriam impossíveis, uma vez que a busca seria realizada em um vetor distinto ao vetor onde o elemento fora incluso.

III. Mutação

Uma das principais diferenças entre os paradigmas de programação funcional e imperativo reside presença de efeitos colaterais (efeitos observáveis, secundários à obtenção de um resultado, que ocorrem durante a execução de uma função). No paradigma funcional, efeitos colaterais são evitados, enquanto que no paradigma imperativo, são abraçados. Essa característica é ainda mais pronunciada em linguagens orientadas à

[3]Um algoritmo de ordenação é considerado estável, quando não muda a ordem original dos elementos caso eles tenham o mesma classificação [9].

objetos, onde a própria definição de objeto embute o conceito de estado.

Vale ressaltar que, apesar da mutabilidade ser abraçada pelo paradigma orientado a objetos, nem todo objeto é obrigatoriamente mutável. Nelson et al. [11] define categorias de mutabilidade baseadas no grau de mutabilidade de um objeto. Segundo essa categorização, objetos podem ser: *imutáveis*, quando não existem mudanças no seu estado interno após a sua construção; terem seu *estado de igualdade imutável*, quando, apesar de existirem mudanças de estado, essas não afetam o estado de igualdade do objeto; e *mutáveis*, quando pode-se alterar indiscriminadamente os atributos de um objeto, sejam eles pertencentes ao seu estado de igualdade ou não.

Apesar de categorizar em sua completude os níveis de impacto de uma mutação no estado de igualdade de um objeto, essa categorização tem uma desvantagem importante: ela é indecidível. Logo, é impossível a escrita de um procedimento algorítmico que verifique o impacto de mutações no estado de igualdade dos objetos nas variadas ramificações de um programa.

Uma categorização menos abrangente, porém decidível, consiste em categorizar objetos em duas categorias distintas: *imutáveis* e *potencialmente mutáveis*. Objetos imutáveis consistem em objetos cuja própria declaração omite construtos que dão suporte à mutabilidade (i.e. é sintaticamente impossível a modificação do estado interno de um objeto). Objetos não categorizados como imutáveis, são, por definição, considerados potencialmente mutáveis.

IV. Mutações e Coleções Baseadas em Igualdade

Uma das possíveis consequências de mutações em um objeto é a alteração do seu estado de igualdade. Essas alterações têm um efeito negativo quando esses elementos estão inseridos em coleções baseadas em igualdade. Pode-se ilustrar esse impacto com o seguinte cenário: dada uma tabela de espalhamento cujo elementos são coordenadas. Mudanças na latitude (ou na longitude) desses elementos podem mudar o seu código *hash*, o que impossibilitaria sua busca, uma vez que a busca poderia ser realizada em um vetor interno distinto ao vetor onde a coordenada fora originalmente incluída.

Algumas técnicas de mitigação foram desenvolvidas no intuito de se mitigar esse problema. Essas técnicas diferem basicamente no grau de mutabilidade dos objetos passíveis de inserção em coleções baseadas em igualdade.

A. Estado de igualdade imutável

A primeira técnica foi proposta por Liskov e Guttag [15] e é usado por Vaziri et al. [16] na definição de tipos de relação – uma extensão da linguagem Java que permite a definição declarativa da igualdade – e consiste na proibição da alteração do estado de igualdade de objetos após a sua construção.

Apesar de ser extremamente restritiva, essa técnica tem como principal vantagem a decidibilidade de sua checagem. De fato, em linguagens como por exemplo C++ e Java, provêm construtos (*const* e *final*) que impedem a alteração de atributos de objetos após a sua inicialização. Nestes casos, tentativas de alteração do valor contido nesses atributos resultam em erros de compilação.

B. Estado de igualdade imutável após inserções

Esta técnica é consiste em só permitir a alteração de informações que impactem o estado de igualdade de um objeto até a sua inserção em uma coleção. Após sua inserção, mutações são proibidas. É usada por Grech et al. [17] em sua proposição de mecanismo para definição de semântica de igualdade em Java.

É interessante notar que essa técnica é suportada nativamente por algumas linguagens de programação, como por exemplo Ruby [18]. Nesta última, todos objetos têm a operação *freeze* – congelar, em inglês. Tentativas de mudança de estado de um objeto subsequentes à invocação dessa operação resulta em exceções em tempo de execução.

Apesar de ser a primeira vista mais flexível que a primeira técnica apresentada, essa técnica apresenta uma desvantagem importante – a sua checagem em tempo de compilação é um problema indecidível. Isso faz com que inconsistências no uso de coleções só sejam descobertas em tempo de execução.

V. Reindexação

Apesar de mitigar o problema em questão, técnicas que impedem a mutabilidade de objetos têm desvantagens importantes. Entre elas a impossibilidade da realização de certas modelagens válidas. Por exemplo, é uma modelagem plausível o armazenamento dos endereços de uma pessoa em um conjunto. Da mesma maneira alterações nesses endereços também são plausíveis. Caso a alteração do estado de igualdade seja proibida, endereços não poderiam ser alterados.

Uma forma de se mitigar essa consiste no emprego de coleções reindexáveis. Coleções reindexáveis diferem-se das coleções convencionais por incorporar uma série de algoritmos auxiliares empregados nos casos onde os elementos não são encontrados através dos procedimentos usuais. Com o uso de coleções reindexáveis pode-se manter a consistência de coleções baseadas na igualdade sem quaisquer restrições à mutabilidade em nenhuma etapa do ciclo de vida de um objeto.

A. Implementação Ingênua

Em termos gerais, a implementação ingênua de uma tabela de espalhamento re-indexável consiste na alteração das operações que façam buscas nos vetores internos da tabela (i.e. operações de busca, inserção e remoção), de modo que, nos casos onde um elemento não é encontrado dentro da tabela pela algoritmo habitual (i.e. baseado na escolha do vetor via código *hash*), seja realizada uma busca secundária, na qual todos os vetores internos da tabela são varridos. Caso o elemento seja encontrado nessa busca secundária, uma operação de reindexação é realizada seguida de uma nova execução da operação original. A reindexação de um elemento consiste na sua remoção seguida pela sua re-inserção. O Algoritmo 1 apresenta uma descrição deste procedimento em pseudo-código.

Algoritmo 1: Reindexação – Implementação ingênua

```
Data: Coleção-alvo c
Data: Operação k a ser executada contra c
Data: Elemento e a ser manipulado por k
Result: Resultado da invocação de k em c
início
    r ← invoca k em c
    se se k for mal-sucedida então
        para cada elemento x na coleção c faça
            se x e e são a mesma instância então
                remove x de c
                adiciona x à c
                retorna resultado de nova execução de k
            fim
        fim
    fim
    retorna r
fim
```

Algoritmo 2: Reindexação – Implementação adaptativa

```
Data: Coleção-alvo c
Data: Operação k a ser executada contra c
Data: Elemento e a ser manipulado por k
Result: Resultado da invocação de k em c
início
    r ← invoca k em c
    se se k for mal-sucedida e e mutável então
        para cada elemento x na coleção c faça
            se x e e são a mesma instância então
                remove x de c
                adiciona x à c
                retorna resultado de nova execução de k
            fim
        fim
    fim
    retorna r
fim
```

A lógica por trás dessa abordagem consiste na ideia de que, independentemente do estado de igualdade (e por consequência seu código *hash*) de um objeto no momento da sua inserção, esse objeto encontra-se em um dos vetores internos da tabela de espalhamento. Por exemplo, digamos que no momento da inserção tinha 1 (um) como valor de código *hash* cujo valor, após uma mutação, passou a ser 3 (três). Apesar da busca convencional ser incapaz de encontrar o objeto (o algoritmo usual apenas procuraria no terceiro vetor, onde não seria encontrado), uma busca linear em todos os vetores seria bem-sucedida, uma vez que ele seria encontrado na varredura do terceiro vetor. Após re-indexação o objeto seria incluído no terceiro vetor e sendo subsequentemente encontrado pelos algoritmos convencionais (a não ser que sofresse uma nova mutação, neste caso o ciclo se repetiria).

B. Implementação com uso de técnicas adaptativas

Segundo Nelson et al. [11], apenas 4% dos elementos têm seu estado de igualdade alterado após sua inclusão em coleções baseadas em igualdade. Sendo assim, conclui-se que o custo inerente à re-indexação muitas vezes não é necessário.

Com o intuito de diminuir esse impacto, pode-se adotar técnicas adaptativas. De acordo com Neto [19], um dispositivo é denominados adaptativo quando seu comportamento muda em função de sua entrada. Através do uso dessas técnicas pode-se incorporar duas otimizações distinta que têm como alvo tanto o algoritmo de reindexação quanto a instanciação condicional das coleções.

O algoritmo adaptativo de reindexação de coleções é semelhante à implementação nativa. Em ambos é realizada uma busca linear seguida por uma reindexação nos casos onde a busca convencional seja infrutífera. Porém, diferentemente da implementação ingênua, a reindexação só é executada quando o elemento sendo manipulado seja potencialmente mutável. Esse comportamento é descrito pelo Algoritmo 2.

Outra otimização compreende a instanciação condicional de coleções reindexáveis. Nesse caso coleções reindexáveis são instanciadas apenas quando forem passíveis de terem elementos potencialmente mutáveis. Para isso, utiliza-se de uma característica inerente às coleções incluídas em bibliotecas[4] encontradas em linguagem fortemente tipadas. Nessas bibliotecas é habitual que coleções possam ter seu tipo refinado através da definição do tipo de seus elementos. Através desse mecanismo, o compilador pode garantir que apenas instâncias de um determinado tipo possam ser incluídas em uma coleção. Dá-se a esse mecanismo o nome de Polimorfismo Paramétrico [1].

Existe porém uma restrição com relação à técnica descrita acima: os tipos usados para parametrizar os elementos de uma coleção podem ser tipos polimórficos (i.e. podem ser estendidos arbitrariamente). Algumas dessas extensões podem eventualmente ter um perfil de mutabilidade distinto. Por exemplo, um subtipo pode definir um novo atributo que porventura seja modificável. Sendo assim, só é possível se aplicar a instanciação condicional de coleções reindexáveis quando o tipo usado para sua parametrizar é imutável e não pode ser ser estendido.

C. Complexidade Algorítmica

Apesar de semelhantes em alguns aspectos, pode-se intuitivamente se inferir que as implementações – ingênua e adaptativa – têm perfis de execução distintos. Logo é desejável uma análise com o intuito de indicar o comportamento de ambos algoritmos, tanto no que tange o tempo de execução – complexidade temporal –, quanto o uso de memória – complexidade espacial. A Tabela I apresenta essas complexidades[5] categorizadas tanto por classe de operação (inclusão, remoção

[4] Pode-se citar, entre outras, o STL em C++, *Java Collections Framework* em Java.

[5] As complexidades referentes à implementação usual de tabela de espalhamento foram obtidas em Cormen et al. [9].

Tabela I: Complexidade Algorítmica – Tabela de Espalhamento

	Convencional[1]		Ingenua		Adaptativa	
Operação	Melhor	Pior	Melhor	Pior	Melhor	Pior
Elementos Mutáveis						
Inserção	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Remoção	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Busca	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Elementos Imutáveis						
Inserção	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(n)$
Remoção	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\boldsymbol{\mathcal{O}(1)}$	$\mathcal{O}(n)$
Busca	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Espacial	$\mathcal{O}(n)$		$\mathcal{O}(n)$		$\mathcal{O}(n)$	

[1] Não trata mutações de elementos após sua inserção.

e busca) quanto pelo perfil dos elementos sendo incluídos (mutáveis e imutáveis). Para a expressão das complexidades foi adotada a notação Grande-O [9].

No que tange a complexidade espacial, não existem diferenças entre tabelas de espalhamentos convencionais e reindexáveis. Porém, no que tange a complexidade temporal, existem diferenças consideráveis. Pode-se notar uma degradação nas operações relativas à inserção e remoção nas coleções reindexáveis. Conforme discutido anteriormente, o processo de reindexação realiza uma busca linear na coleção nos casos onde os elementos não são encontrados. Buscas lineares têm a complexidade temporal $\mathcal{O}(n)$. Logo, a complexidade temporal das inserções e remoções degrada para $\mathcal{O}(n)$.

Notavelmente, a implementação baseada em técnicas adaptativas só apresenta degradação quando os elementos sendo inseridos (ou removidos) na coleção são mutáveis. No caso dos elementos imutáveis, essa implementação comporta-se de maneira semelhante à implementação convencional.

VI. Protótipo

Apesar da análise da complexidade algorítmica através da notação Grande-O ser capaz expressar do comportamento dos algoritmos com relação ao tamanho de sua entrada, não se pode através dela se inferir o tempo absoluto gasto por um algoritmo. Por exemplo, dois algoritmos que realizam uma operação para cada elemento de sua entrada têm a mesma complexidade temporal – $\mathcal{O}(n)$) –, independentemente do tempo gasto por cada um nessas operações. Tendo em vista esta limitação, foi elaborado um experimento para a medição dos impactos de ambas as técnicas – ingênua e adaptativa – no desempenho geral de programas. Para isso é necessária a implementação de um protótipo de ambas soluções.

Vale-se notar que não foi realizado nenhuma tentativa de otimização das estruturas e algoritmos aqui utilizadas, uma vez que o objetivo principal deste artigo não é a definição de uma implementação de coleções reindexáveis plenamente otimizadas e sim o estudo do impacto relativo do uso de coleções reindexáveis no desempenho. É importante salientar também que a implementação não é estruturalmente ótima: o uso de decoradores [4] para a inclusão das operações de reindexação levou ao uso de técnicas de programação reflexiva, que por sua vez acarretou a quebra do encapsulamento. O paradigma da linguagem de programação também influiu na estrutura adotada pelo protótipo. É possível, por exemplo, se realizar uma implementação mais elegante (e eficiente) em linguagens baseadas em protótipos (como por exemplo SELF e Javascript [20], [21]). Nessas linguagens é possível a manipulação da hierarquia de um objeto em tempo de execução. Logo, pode-se alterar a implementação de coleção utilizada no momento da inserção (ou remoção) de um elemento.

A. Detalhes de implementação

É desejável que o protótipo seja capaz de substituir as implementações de coleções por suas contrapartidas reindexáveis sem que seja necessária a alteração dos programas, uma vez que esta pode introduzir novos defeitos. Uma solução eficaz para esse problema reside na adoção de técnicas da programação orientada a aspectos [22]. Vislumbrado por Gregor Kiczales, esse paradigma tem como mote a decomposição de problemas em interesses transversais – funcionalidades que não são pertencem à um construto específico e são utilizadas de maneira dispersa por todo código. Através desse paradigma, pode-se definir comportamentos adicionais (denominados conselhos ou *advices*) a serem injetados em pontos específicos de um programa (chamados de pontos de corte ou *pointcuts*).

Pode-se aplicar essa capacidade de definição de comportamento alternativos para injetar as operações referentes à reindexação nas coleções baseadas em igualdade. Apesar da linguagem Java não ter suporte nativo à programação orientada a aspectos, é possível a incorporação desse paradigma através de bibliotecas como o AspectJ, essa última utilizada na modalidade de inclusão de aspectos em tempo de execução (*load-time weaving*). Essa modalidade tem como vantagem dispensar a recompilação dos programas usados pela suíte de testes, porém não permite a instrumentalização de classes pertencentes ao JDK (*Java Development Kit*). Em razão dessa limitação, optou-se pela interceptação de suas chamadas de instanciação ao contrário da modificação das classes de coleções propriamente ditas.

Ao invés de se realizar uma implementação uma implementação *clean room* (i.e. uma nova implementação, sem dependências), optou-se por adotar uma abordagem baseada em decoradores [4]. Nessa abordagem, as funcionalidades básicas são supridas pela implementação convencional das coleções e as operações passíveis de reindexação são decoradas de modo a realizá-la quando necessário. A principal razão para a adoção desse procedimento reside na maior fidelidade nas medições, uma vez que uma nova implementação poderia não ter o mesmo grau de otimização das coleções nativas.

B. Instanciação condicional de coleções

Conforme discutido anteriormente, a partir da versão 1.5, a biblioteca de coleções da linguagem Java dá suporte ao polimorfismo paramétrico. A primeira vista, pode-se usar essas informações para se determinar o grau de mutabilidade dos

elementos de uma coleção. E, a partir dessa informação, se determinar se é necessária uma implementação re-indexável.

Essa informação porém é removida do código-objeto durante a compilação através de um processo chamado apagamento de tipos (*Type erasure*, em inglês), cuja motivação é a manutenção da compatibilidade do código-objeto em diferentes versões da plataforma Java [3].

De modo a simular a presença das informações removidas através do processo de apagamento de tipos, optou-se por disponibilizar o código-fonte dos programas sendo testados no *classpath*[6] da suíte de testes. No momento da instanciação de uma coleção, a linha correspondente no código-fonte é analisada. Caso o seus elementos sejam instâncias de uma classe considerada imutável (i.e. todos seus atributos são declarados como finais e têm tipos imutáveis, e estende uma classe imutável; ou é uma das classes previamente assim categorizadas[7]), é utilizada a implementação de coleção convencional, caso contrário, é utilizada uma implementação re-indexável. O Algoritmo 3 descreve o procedimento de decisão sobre a imutabilidade de uma classe.

Algoritmo 3: Procedimento de Checagem de Mutabilidade

```
Data: Classe c, cuja mutabilidade se deseja verificar
Result: verdadeiro caso a classe c seja potencialmente
        mutável, falso caso contrário
início
    se se c for uma classes imutável por definição então
        return falso
    senão se a super-classe c for imutável então
        para cada campo a da classe c faça
            se classe de a for p. mutável então
                retorna verdadeiro
            fim
        fim
        retorna falso
    senão
        retorna verdadeiro
    fim
fim
```

Vale destacar que o mecanismo utilizado para contornar a limitação imposta pelo apagamento de tipos pode ser impraticável em ambiente produtivo. Além de ter um alto custo computacional, há a necessidade da inclusão do código-fonte, o que pode ser um empecilho nos casos onde, por razões de segurança, a ofuscação do código seja desejável.

VII. Experimento

Conforme previamente discutido, somente a análise de complexidade temporal de um algoritmo não é suficiente para a comparação do impacto no desempenho (e espaço) da adoção das técnicas apresentadas. Para isso é necessária a condução de um experimento, no qual o impacto da adoção das técnicas no desempenho de programas de variados segmentos é analisado.

No intuito de se obter resultados mais fidedignos aos resultados obtidos pelo uso das técnicas, foi utilizada a versão 9.12-bach da suíte de testes DaCapo [10] . Essa suíte de testes é composta por uma gama variada de programas de código-fonte aberto, entre os quais encontram-se compiladores, renderizadores e gerenciadores de banco de dados. Essa abrangência faz com que os resultados obtidos sejam próximos aos impactos esperados em cenários do mundo-real.

A. Métricas

Para a medição dos impactos no desempenho foram definidas duas métricas distintas: o tempo de execução e a pressão de memória. O tempo de execução (medido em segundos) é uma métrica de simples entendimento que consiste no tempo necessário para a execução de um número determinado de operações (cuja semântica depende do teste de desempenho sendo executado). Já pressão de memória tem como motivação indicar o grau de utilização da memória durante o cenário de testes.

Diferentemente de linguagens como C++, na linguagem Java a memória utilizada pelos programas é reciclada de forma automática, através de um componente denominado Coletor de Lixo [23]. De forma geral, um Coletor de Lixo varre a memória e libera blocos que não sejam mais utilizados. Conforme as características de uso de memória – taxa de alocação e tempo de vida de objetos – o coletor de lixo pode ser mais ou menos demandado.

$$\mathcal{P}_x = \frac{\tau_x}{\tau_0} \tag{1}$$

A Fórmula 1 calcula a pressão de memória $\mathcal{P}_x$ de uma execução x, onde τ_x e τ_0 são respectivamente a quantidade de memória manipulada pelo processo de coleta de lixo na execução x e da linha base. Por definição, a pressão de memória da linha base é 1. Notavelmente, a pressão de memória deve ser interpretada como uma métrica de comparação entre duas execuções distintas: quanto maior seu valor, mais o coletor de lixo foi exigido e consequente mais do subsistema de memória foi demandado.

B. Método

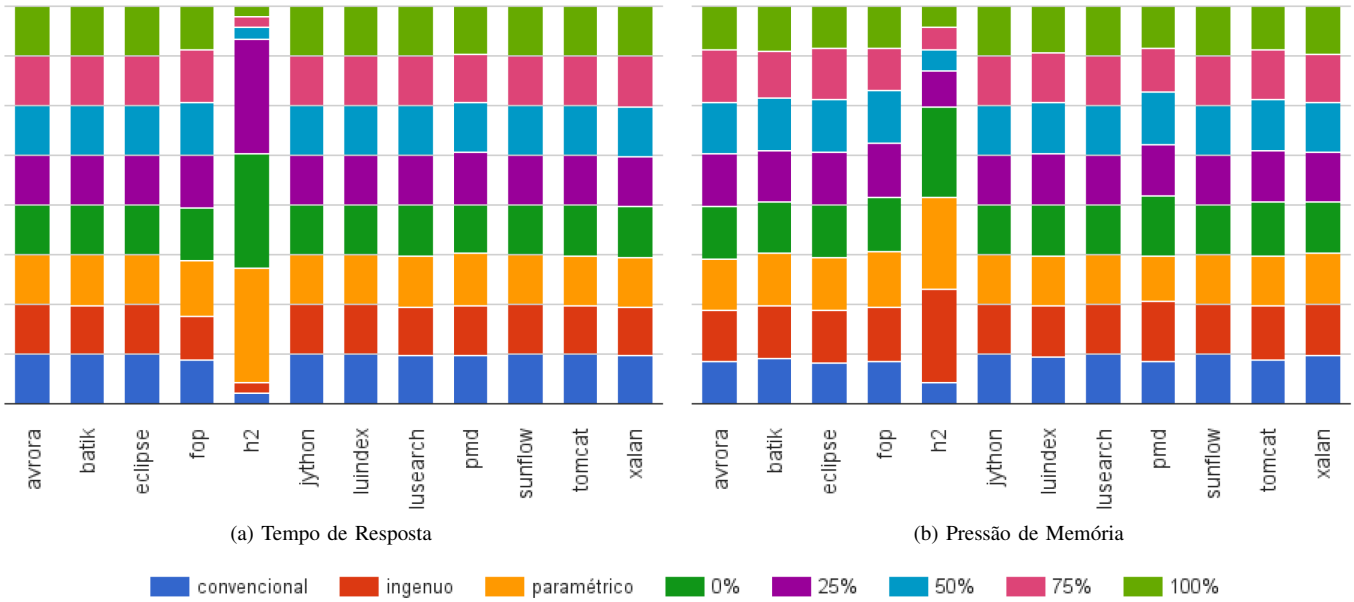
É importante porém ressaltar que a suíte de testes tivera que ser modificado[8] durante a realização do experimento. Essas alterações compreendem: a atualização de programas – *Apache Tomcat* – incompatíveis com a versão do JDK utilizada; a disponibilização do código-fonte dos programas analisados no *classpath*; e a inclusão do protótipo propriamente dito. Além disso, alguns programas – *tradebeans* e *tradesoap* – foram removidos do experimento. Esses programas são baseados do servidor de aplicação Apache Geronimo[9], que utiliza-se de

[6]Conjunto de recursos que podem ser carregados pela JVM. Uma classe deve estar incluída no *classpath* para ser utilizada.

[7]Enumerados; todos os tipos primitivos e suas classes correspondentes; e *String*, *BigInteger*, *BigDecimal* e *Object*.

[8]O pacote modificado, acrescido do código-fonte do protótipo, encontra-se disponível em https://drive.google.com/open?id=0B4-smhbrEJAaRGI1NXdJVUtKbGM.

[9]Disponível em http://geronimo.apache.org/.



(a) Tempo de Resposta

(b) Pressão de Memória

Figura 1: Resultados do Experimento

um mecanismo de carregamento de classe incompatível com instrumentação de código utilizado pelo AspectJ.

Inicialmente foram definidos três cenários de medição, referentes à linha-base, à implementação ingênua e a implementação adaptativa. Estudos realizados por Parnin et al. [24] porém mostraram um padrão de utilização de coleções parametrizadas que poderia comprometer o experimento. Apesar da utilização em código mais recentes de coleções parametrizadas, não existe um movimento de modernização de bases de código mais antigas para a sua adoção. Esse cenário invalidaria as medições referentes aos ganhos do uso de técnicas adaptativas, uma vez que nesse caso a última se comportaria de maneira semelhante à implementação ingênua. Tendo em vista essa conjuntura, foram definidos outros cinco cenários baseados na implementação adaptativa, nos quais a decisão sobre o uso de coleções reindexáveis é baseada em razões (0%, 25%, 50%, 75% e 100%) previamente definidas.

Notavelmente, alguns cenários executam procedimentos adicionais que pode ter um impacto considerável nas medições. Tendo em vista que o objetivo principal é o estudo do impacto das técnicas na execução de programas, se fez necessária uma normalização desses resultados. Para isso, todos os cenários executam as mesmas operações auxiliares (procedimentos de inferência sobre a mutabilidade de elementos de uma coleção e a obtenção de números randômicos para os cenários baseados na razão de objetos imutáveis são sempre realizados), independentemente da utilização de seus resultados. Um efeito colateral dessa abordagem é a disparidade entre as medidas obtidas pela linha base e pela execução da suíte de teste sem modificações.

Além das métricas de desempenho definidas anteriormente, também foram colhidos dados referente ao perfil de utilização de coleções em cada um dos programas utilizados. Esses dados englobam à implementação utilizadas – convencional ou reindexável – bem como a quantidade de instâncias criadas. A principal motivação para a obtenção desses dados auxiliares é se traçar um perfil de uso de coleções em cada um dos cenários estudados.

Tabela II: Perfil de Uso de Coleções

Programa	Instâncias Convencionais	Instâncias Reindexáveis	Instâncias Totais	%
avrora	0	469	469	100%
batik	0	2300	2300	100%
eclipse	0	2178	2178	100%
fop	0	11554	11554	100%
h2	0	8704171	8704171	100%
luindex	0	77	77	100%
lusearch	0	1156	1156	100%
pmd	203217	25195	228412	11,3%
tomcat	13309	10249	23684	43,51%
xalan	0	110338	110338	100%
jython[1]	–	–	–	–
sunflow[2]	–	–	–	–
tradebeans[3]	–	–	–	–
tradesoap[4]	–	–	–	–

[1, 2] Não foram criadas coleções durante a execução.
[3, 4] Baseados no Apache Geronimo, cujo mecanismo de carga de classe impediu a utilização do dispositivo de injeção de aspectos em tempo de execução.

VIII. Resultados

Tendo em vista o método e protótipo previamente descritos, foram executadas medições que encontram-se sumarizadas na

Tabela III: Resultados Obtidos

Programa	Linha Base	Ingenua		Paramétrica		Adaptativa 0%		25%		50%		75%		100%	
avrora	11,06s	11,05s	1,20	11,08s	1,20	11,08s	1,24	11,07s	1,22	11,08s	1,21	11,03s	1,24	11,05s	1,01
batik	2,04s	2,07s	1,14	2,07s	1,14	2,06s	1,14	2,06s	1,12	2,06s	1,12	2,05s	1,02	2,03s	1,00
eclipse	93,08s	93,33s	1,27	92,97s	1,26	92,64s	1,26	92,66s	1,27	92,67s	1,27	92,89s	1,27	93,36s	1,00
fop	0,27s	0,34s	1,32	0,33s	1,31	0,32s	1,31	0,33s	1,30	0,32s	1,28	0,27s	1,01	0,27s	1,00
h2	26,16s	274,00s	4,31	276,77s	4,27	275,01s	4,21	27,42s	1,70	26,07s	1,00	26,20s	1,00	26,31s	1,00
jython	8,58s	8,48s	1,00	8,56s	1,00	8,74s	1,00	8,55s	1,00	8,52s	1,00	8,55s	1,00	8,53s	1,00
luindex	0,90s	0,90s	1,09	0,90s	1,09	0,90s	1,09	0,89s	1,09	0,89s	1,09	0,90s	1,08	0,90s	1,00
lusearch	1,11s	1,15s	1,00	1,15s	1,00	1,14s	1,00	1,13s	1,00	1,13s	1,00	1,12s	1,00	1,08s	1,00
pmd	1,53s	1,58s	1,39	1,52s	1,06	1,61s	1,39	1,51s	1,22	1,51s	1,20	1,47s	1,02	1,50s	1,00
sunflow	3,77s	3,80s	1,00	3,78s	1,00	3,77s	1,00	3,78s	1,00	3,77s	1,00	3,78s	1,00	3,81s	1,00
tomcat	5,60s	5,72s	1,27	5,67s	1,11	5,73s	1,27	5,69s	1,19	5,69s	1,16	5,62s	1,15	5,58s	1,00
xalan	4,40s	4,57s	1,07	4,59s	1,07	4,59s	1,06	4,56s	1,05	4,60s	1,03	4,54s	1,01	4,42s	1,00
tradebeans[1]	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
tradesoap[2]	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

Ambiente de execução: i7-4790K (3,6 Ghz) com 32GB de RAM rodando Windows 10. Foi usado a JDK 1.8.0_92 (64 bits), sem nenhum qualquer tipo de ajustes dos parâmetros de execução (i.e. tamanho da memória *heap* e parâmetros do coletor de lixo).
[1, 2] Baseados no Apache Geronimo, cujo mecanismo de carga de classe impediu a utilização do dispositivo de injeção de aspectos em tempo de execução.

Tabela III e no Gráfico 1 (resultados detalhados estão dispostos no Apêndice A). No intuito de se diminuir as desvios, cada cenário foi executado cem vezes com a obtenção da mediana dos valores.

Pode-se a primeira vista notar-se impactos mais pronunciados nas medidas obtidas pelo *H2*, onde os tempos de processamento decorrido na execução implementação ingênua foi cerca onze vezes maior que a linha-base. De maneira recíproca, alguns programas – *sunflow* e *jython* – não tiveram diferenças significativas tanto no tempo de processamento quanto na pressão memória através dos cenários. Essa discrepância está relacionada com o perfil de uso (disposto na Tabela II) das coleções em cada um dos programas analisados. Pode-se notar uma correlação entre o desempenho do programa e a quantidade de coleções reindexáveis criadas. Programas com maior quantidade de coleções criadas – *h2*, *pmd* e *tomcat* – estão entre os programas que tiveram maior impacto em seu desempenho (*h2*, *pmd*, *fop* e *tomcat*).

Apesar de comparativamente não criar muitas instâncias, o programa *fop*[10] obteve uma queda mais expressiva de desempenho que o *tomcat*, que ocupa o terceiro lugar na classificação de programas por número de coleções criadas. Pode-se intuitivamente se atribuir essa discrepância a dois fatores. O primeiro fator relaciona as diferenças no modo de execução dos dois cenários: enquanto o primeiro realiza suas operações basicamente dentro de um único processo, o último é baseado em requisições enviadas através do endereço de *loopback* por outro processo. Supõe-se que o tempo gasto em operações relativas à comunicação via rede dilua o impacto no desempenho no último cenário.

Já o segundo fator refere-se a presença de ruídos de medição: o tempo de execução do cenário baseado no *fop* é ordens de magnitude menor que o do baseado no *tomcat*, o que faz com que ruídos nas medições se tornem mais pronunciados.

Também pode-se notar que, em linhas gerais, a implementação adaptativa baseada nas informações de tipo das coleções não obteve um desempenho muito melhor da implementação ingênua. Essa observação corrobora a observação de Parnin et al. [24] de que programas mais antigos não sofreram alterações para o uso de coleções parametrizadas.

O estudo do perfil das coleções indicou que apenas dois programas – *tomcat* e *pmd* – se beneficiaram da técnica adaptativa. Nesses dois casos, os resultados obtidos através da instanciação condicional se equipararam aos resultados obtidos através dos cenários onde as razão entre coleções convencionais e reindexáveis eram pré-definidas (no caso do *tomcat*, os resultados obtidos pela instanciação condicional[11] e paramétrica foram 5,67s e 5,69s respectivamente, enquanto no *pmd* foram 1,52s e 1,50s).

Os resultados obtidos mostram que o emprego de técnicas adaptativa aliado ao uso de anotações de tipos em coleções pode mitigar os impactos no desempenho associados ao uso de coleções indexáveis. Sendo por meio destes possível a obtenção de resultados semelhantes aos obtidos através da implementação convencional (i.e. sem reindexação).

IX. Trabalhos Correlatos

Nelson et al. [11] analisa o impacto da mutabilidade de elementos em coleções. Suas conclusões indicam que a grande maioria dos objetos inseridos em coleções baseadas em igualdade não mudam seu estado de igualdade, mesmo quando são mutáveis. Esse cenário muda quando coleções não-baseadas na igualdade são levadas em conta. Nesse caso, mais da metade dos objetos apresentam mudanças após serem inseridos

[10] Um conversor da linguagem de marcação XSL-FO para PDF. Disponível em https://xmlgraphics.apache.org/fop/.

[11] Utilizada os cenários cujas razões são 50% e 100% para o*tomcat* e o *pmd*, respectivamente.

em coleções. Isto indica um potencial problema, uma vez que implementação dessas coleções pode ser substituída por implementações baseadas na igualdade sem qualquer tipo de aviso por parte do compilador.

Conforme previamente discutido, uma outra possibilidade de mitigação do problema apresentado consiste na refatoração das bibliotecas de coleções de modo que o sistema de tipos impeça a inclusão de objetos potencialmente mutáveis em coleções baseadas em igualdade. Para isso, se faz necessário o suporte a qualificadores de tipos. Existe na Literatura uma série de proposições [25], [26] de extensões da linguagem Java cujo sistema de tipos prevê suporte a qualificadores de tipos. Ainda nessa linha, uma abordagem menos intrusiva compreende a adoção de um sistema de tipos plugável [5] como o *Checker Framework* [6], [7]. Vale citar que que o último é disponibilizado com uma implementação de qualificadores de tipos que tratam da imutabilidade.

Pode-se também citar como trabalho correlato a ferramenta Chameleon [27], que tem como principal motivação a decisão da melhor implementação de coleções a ser utilizada. Porém, ao contrário do estudo aqui apresentado, essa ferramenta não altera as implementações em tempo de execução. Ao invés disso, apresenta um relatório onde sugere mudanças relativas às implementações utilizadas, ficando a cargo do desenvolvedor acatar as sugestões.

X. Trabalhos Futuros

Durante a execução desta pesquisa foram identificados alguns possíveis desdobramentos. Um desses desdobramentos consiste na realização de novos testes quando a reificação de tipos genéricos for suportada pelo Java. A inclusão do suporte à reificação de tipos está prevista para a versão 10 da linguagem Java, através do Projeto *Valhalla*[12].

Um outro possível desmembramento consiste num estudo exploratório sobre os impactos da adoção de uma biblioteca de coleções estruturada de modo a eliminar a possibilidade da inclusão de elementos mutáveis em coleções baseadas em igualdade é utilizada. Apesar da difusão do uso de uma biblioteca moldes ser improvável, os resultados obtidos através dessa pesquisa serão úteis no desenvolvimento de bibliotecas para novas linguagens.

Uma ramificação viável deste trabalho consiste em um estudo exploratório do impacto relativo ao uso instanciação condicional de coleções incorporando otimizações específicas para certos tipos de elementos no desempenho. Pode-se citar com exemplo de tipos que podem se beneficiar de implementações alternativa de coleções são os enumerados – tipo de dados representando um domínio de valores válidos. Uma característica importante dos enumerados é a possibilidade de serem mapeados para inteiros, o que permite a simulação de tabela de espalhamento através de um simples vetor[13]. Seguindo essa linha, existem implementações alternativas de coleções[14] já otimizadas para o armazenamento de tipos primitivos.

XI. Conclusão

Apesar de relativamente incomum, mutações em elementos de coleções baseadas em igualdade é problemática, uma vez que impossibilita a busca destes através dos algoritmos de busca convencionais.

A solução conceitualmente mais adequada para esse problema consiste na formalização de dois tipos distintos de coleções: coleções não-baseadas e baseadas em igualdade. O sistema de tipos deve garantir que apenas objetos imutáveis sejam inclusos na última. Essa solução porém se mostra impraticável no cenário atual, uma vez que sua adoção acarretaria uma série de incompatibilidades, tanto no nível de código-fonte quanto de código-objeto.

Outra forma de solucionar-se o problema consiste na criação de coleções reindexáveis (i.e. coleções passíveis cuja consistência é mantida mesmo quando um elemento sofre mutações). Uma implementação ingênua desse conceito consiste em implementações alternativas de coleções baseadas em igualdade, onde todos os objetos são tratados como mutáveis e buscas mal-sucedidas são seguidas de uma operação de reindexação.

Resultados obtidos experimentalmente mostram uma sensível queda de desempenho quando a técnica ingênua é empregada. Com adoção de técnicas adaptativas pode-se mitigar esses efeitos negativos, sendo possível a eliminação total das perdas quando todos os elementos inseridos em coleções baseadas em igualdade são imutáveis. Contudo, algumas questões impedem uso dessa técnica de maneira abrangente.

A primeira dificuldade é relativa a ausência de anotações referentes ao tipo de elementos de coleções na maioria dos programas analisados. Essa observação é corroborada por Parnin et al. [24], que constata que apesar de terem sido amplamente adotada desde a sua incepção, é relativamente rara a migração de código já existente unicamente para a inclusão dessas parametrizações.

Outra dificuldade é relacionada com a falta de um mecanismo de reificação das parametrizações de coleções na versão atual da linguagem Java. No experimento esse dispositivo foi simulado através de uma implementação baseada no processamento do código-fonte, que deve estar disponível em tempo de execução. Apesar de eficaz, essa implementação não é adequada no ambiente produtivo, uma vez o processo de análise do código fonte é relativamente custoso e a presença do código-fonte pode ser muitas vezes proibida por questões de segurança.

Sumarizando, técnicas adaptativas podem diminuir consideravelmente o custo do uso de coleções reindexáveis. Porém, para a construção de uma solução robusta, é necessário que tanto a Máquina Virtual Java de suporte a reificação de tipos de parâmetros e incorporação de coleções reindexáveis ao Kit de Desenvolvimento Java, uma vez que a instrumentalização das classes do último é problemática.

[12]Disponível em http://openjdk.java.net/projects/valhalla/.

[13]O JDK inclui uma implementação de tabela de espalhamento (*java.util.EnumMap*) seguindo esses moldes.

[14]Como por exemplo o GNU Trove. Disponível em http://trove.starlight-systems.com/.

REFERÊNCIAS

[1] PIERCE, B. C. *Types and Programming Languages*. 1st. ed. Cambridge, MA, USA: The MIT Press, 2002. ISBN 0262162091, 9780262162098.

[2] LISKOV, B. H.; WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 16, n. 6, p. 1811–1841, nov. 1994. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/197320.197383>.

[3] NAFTALIN, M.; WADLER, P. *Java generics and collections*. Sebastopol, CA, EUA: O'Reilly Media, Inc., 2007.

[4] GAMMA, E. et al. *Design patterns: elements of*. Upper Saddle River, NJ, USA: Addison-Wesley, 1994.

[5] BRACHA, G. Pluggable type systems. In: CITESEER. *OOPSLA workshop on revival of dynamic languages*. Vancouver, BC, Canada, 2004. v. 1.

[6] PAPI, M. M. et al. Practical pluggable types for java. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2008. (ISSTA '08), p. 201–212. ISBN 978-1-60558-050-0. Disponível em: <http://doi.acm.org/10.1145/1390630.1390656>.

[7] DIETL, W. et al. Building and using pluggable type-checkers. In: *Proceedings of the 33rd International Conference on Software Engineering*. New York, NY, USA: ACM, 2011. (ICSE '11), p. 681–690. ISBN 978-1-4503-0445-0. Disponível em: <http://doi.acm.org/10.1145/1985793.1985889>.

[8] DRAFT, W. Standard for programming language c++. *ISO/IEC N*, v. 4296, p. 2014, 2009.

[9] CORMEN, T. H. et al. *Introduction to algorithms*. Cambridge, MA, USA: MIT press Cambridge, 2001. v. 6.

[10] BLACKBURN, S. M. et al. The DaCapo benchmarks: Java benchmarking development and analysis. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, 2006. p. 169–190.

[11] NELSON, S.; PEARCE, D. J.; NOBLE, J. Understanding the impact of collection contracts on design. In: VITEK, J. (Ed.). *Objects, Models, Components, Patterns: 48th International Conference, TOOLS 2010, Málaga, Spain, June 28–July 2, 2010. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. p. 61–78. ISBN 978-3-642-13953-6. Disponível em: <http://dx.doi.org/10.1007/978-3-642-13953-6_4>.

[12] TARSKI, A. *Introduction to Logic and to the Methodology of the Deductive Sciences (Oxford Logic Guides)*. Oxford: Oxford University Press, 1994. ISBN 019504472X.

[13] GÖDEL, K. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für Mathematik und Physik*, v. 38, n. 1, p. 173–198, 1931. ISSN 1436-5081. Disponível em: <http://dx.doi.org/10.1007/BF01700692>.

[14] KONHEIM, A. G. *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. Hoboken, NJ, USA: John Wiley & Sons, 2010.

[15] LISKOV, B.; GUTTAG, J. *Abstraction and specification in program development*. Cambridge, MA, USA: MIT press, 1986.

[16] VAZIRI, M. et al. Declarative object identity using relation types. In: ERNST, E. (Ed.). *ECOOP 2007 – Object-Oriented Programming: 21st European Conference, Berlin, Germany, July 30 - August 3, 2007. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 54–78. ISBN 978-3-540-73589-2. Disponível em: <http://dx.doi.org/10.1007/978-3-540-73589-2_4>.

[17] GRECH, N.; RATHKE, J.; FISCHER, B. Jequalitygen: Generating equality and hashing methods. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2010. (GPCE '10), p. 177–186. ISBN 978-1-4503-0154-1. Disponível em: <http://doi.acm.org/10.1145/1868294.1868320>.

[18] FLANAGAN, D.; MATSUMOTO, Y. *The ruby programming language*. Sebastopol, CA, EUA: O'Reilly Media, Inc., 2008.

[19] NETO, J. J. Adaptive rule-driven devices - general formulation and case study. In: _____. *Implementation and Application of Automata: 6th International Conference, CIAA 2001 Pretoria, South Africa, July 23–25, 2001 Revised Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. p. 234–250. ISBN 978-3-540-36390-3. Disponível em: <http://dx.doi.org/10.1007/3-540-36390-4_20>.

[20] UNGAR, D.; SMITH, R. B. Self: The power of simplicity. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 22, n. 12, p. 227–242, dez. 1987. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/38807.38828>.

[21] FLANAGAN, D. *JavaScript: the definitive guide*. Sebastopol, CA, EUA: O'Reilly Media, Inc., 2006.

[22] KICZALES, G. et al. Aspect-oriented programming. In: AKŞIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 220–242. ISBN 978-3-540-69127-3. Disponível em: <http://dx.doi.org/10.1007/BFb0053381>.

[23] JONES, R.; HOSKING, A.; MOSS, E. *The garbage collection handbook: the art of automatic memory management*. London: Chapman & Hall/CRC, 2011.

[24] PARNIN, C.; BIRD, C.; MURPHY-HILL, E. Adoption and use of java generics. *Empirical Software Engineering*, v. 18, n. 6, p. 1047–1089, 2013. ISSN 1573-7616. Disponível em: <http://dx.doi.org/10.1007/s10664-012-9236-6>.

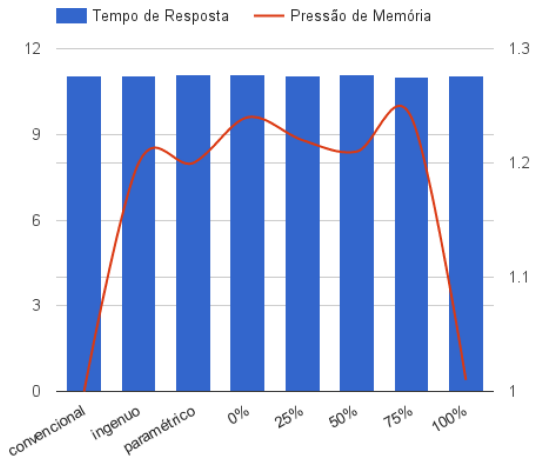
[25] ZIBIN, Y. et al. Object and reference immutability using java generics. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. New York, NY, USA: ACM, 2007. (ESEC-FSE '07), p. 75–84. ISBN 978-1-59593-811-4. Disponível em: <http://doi.acm.org/10.1145/1287624.1287637>.

[26] ZIBIN, Y. et al. Ownership and immutability in generic java. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 45, n. 10, p. 598–617, out. 2010. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1932682.1869509>.

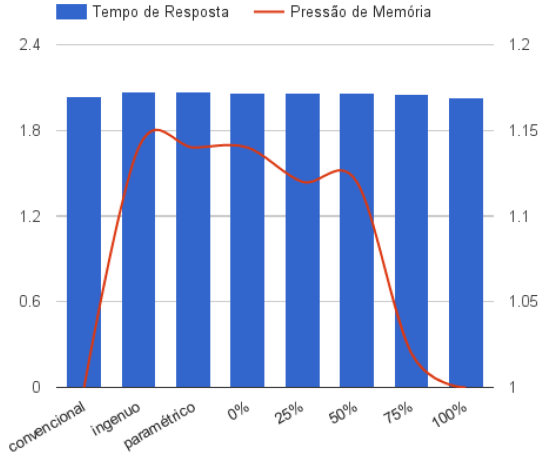
[27] SHACHAM, O.; VECHEV, M.; YAHAV, E. Chameleon: Adaptive selection of collections. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 44, n. 6, p. 408–418, jun. 2009. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1543135.1542522>.

APÊNDICE A
DETALHAMENTO DOS RESULTADOS DO EXPERIMENTO

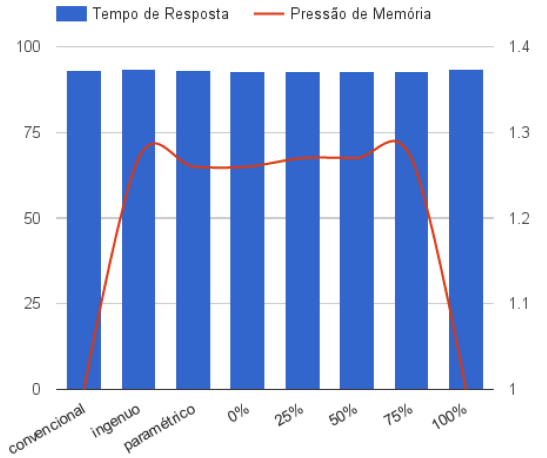
A. *avrora*



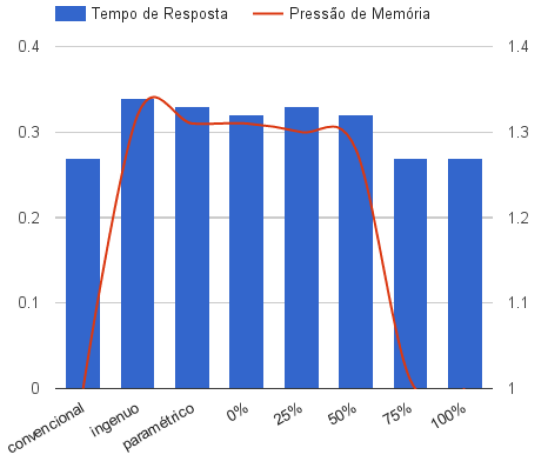
B. *batik*



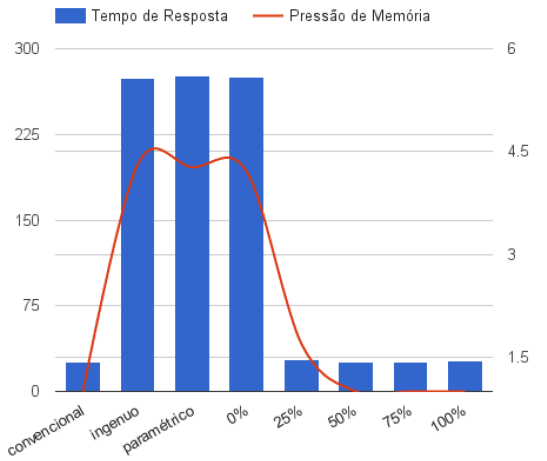
C. *eclipse*



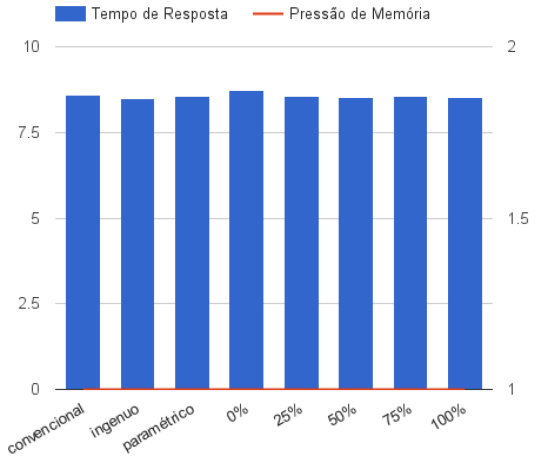
D. *fop*



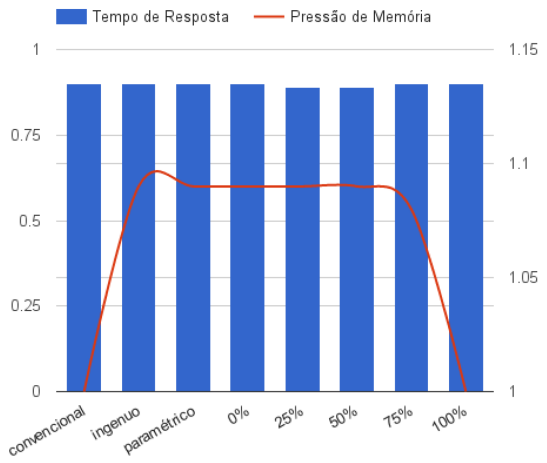
E. *h2*



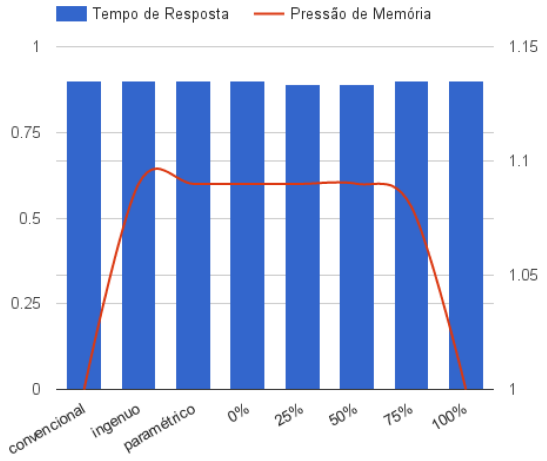
F. *jython*



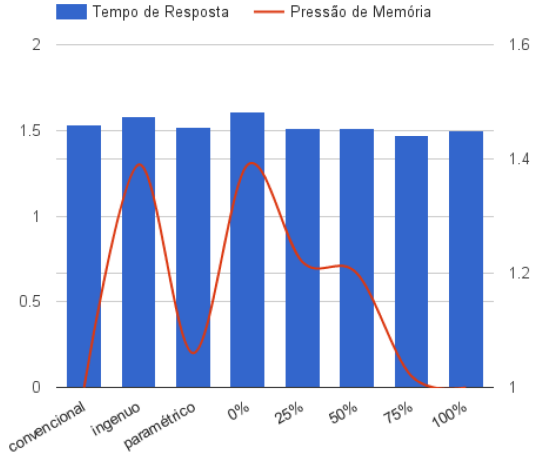
G. luindex



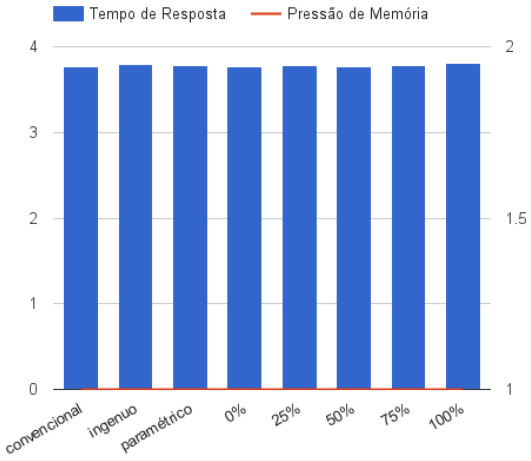
H. lusearch



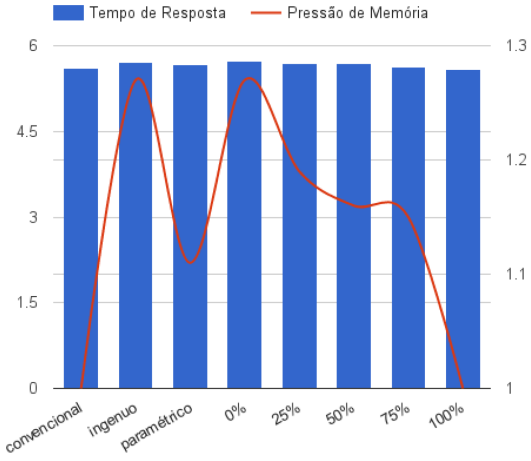
I. pmd



J. sunflow



K. tomcat



L. xalan