1
Introdução

O gerenciamento dinâmico de memória continua sendo uma das áreas de pesquisa mais ativas dentro da Ciência da Computação. Esta busca por novas soluções tem como objetivo não apenas atender à necessidade óbvia de administrar de forma mais eficiente um recurso crítico para o desempenho de sistemas de computação tradicionais, mas também suprir as demandas inerentes ao desenvolvimento acelerado de dispositivos inteligentes cada vez mais sofisticados e heterodoxos.

A despeito do tempo de processamento adicional necessário à execução de rotinas dedicadas ao gerenciamento automático de memória, linguagens de programação que empregam coletores de lixo vêm tornando-se cada vez mais comuns[1]. Diferentes motivos ajudam a explicar tal tendência, dentre os quais destacam-se:

- O gerenciamento automático elimina totalmente os complexos erros relacionados à dinâmica de desalocação explícita de memória, como ponteiros quebrados (dangling pointers) e vazamentos de memória (memory leaks), proporcionando expressivos ganhos de produtividade. Rovner [57] por exemplo sugere que 40% do tempo dispendido no desenvolvimento do sistema Mesa foi dedicado a solução de problemas relacionados ao gerenciamento explícito de memória, sobretudo à correção de erros.

- Quando diferentes componentes de um sistema atuam sobre estruturas de dados comuns, torna-se necessário manter um mecanismo de coordenação global para definir quem deve efetuar a desalocação dos recursos compartilhados, e quando a desalocação deve ser efetuada. O gerenciamento automático de memória evita essa interdependência, possibilitando uma maior modularidade do código.

[1]Das dez linguagens de programação mais populares segundo o índice TIOBE [10], apenas C, C++ e Delphi exigem o gerenciamento explícito da memória. Mesmo no caso destas três linguagens existem diversas implementações que oferecem coletores de lixo (por exemplo, [19, 14, 17, 56]).

PUC-Rio - Certificação Digital Nº 0310845/CA

- Muitas vezes é impraticável, ou mesmo impossível, um programa conseguir determinar se uma estrutura de dados qualquer ainda será acessada a partir de um dado ponto de execução. Este problema é particularmente crítico em linguagens cujo fluxo de execução pode ser modificado através de abstrações sofisticadas como continuações e fechamentos (function closures). Como coletores de lixo geralmente tem acesso irrestrito às estruturas internas mantidas pelo ambiente de execução, ou seja, possuem mais informação, eles são capazes de maximizar a reciclagem da memória mesmo diante de fluxos de execução não-determinísticos e relativamente complexos.

- O crescente hiato entre a velocidade de processamento de CPUs e de circuitos de memória vem tornando a distribuição física de objetos na memória um fator cada vez mais relevante para o desempenho de sistemas de computação. Diversos algoritmos empregados por coletores de lixo movem fisicamente os objetos usados pela aplicação, criando uma distribuição mais adequada ao seu padrão de acesso (localidade). Em alguns casos, conforme comprovam diferentes estudos [71, 55, 15, 37], os ganhos de desempenho decorrentes desta maior localidade fazem com que o uso de coletores de lixo efetivamente melhore a velocidade de execução da aplicação.

O gerenciamento automático de memória aumenta o nível de abstração de uma linguagem de programação sem necessariamente modificar a sua semântica básica. Entretanto, por vezes pode ser interessante permitir que um *programa cliente* interaja diretamente com o coletor de lixo, quer seja alterando o seu comportamento, ou simplesmente obtendo informações geradas a partir da sua execução[2]. Como exemplo, informações sobre a conectividade de objetos, geradas através do processo de rastreamento de lixo, podem ser usadas diretamente pelo programa cliente para gerenciar estruturas de dados complexas.

Buscando portanto uma maior flexibilidade, muitas linguagens oferecem mecanismos que possibilitam a interação dinâmica entre programas clientes e o coletor de lixo. Tipicamente, esses mecanismos incluem *finalizadores*, *referências fracas* e métodos para a invocação explícita e parametrização dinâmica do coletor de lixo.

Finalizadores são rotinas invocadas automaticamente pelo coletor de lixo antes de ele liberar a memória ocupada por um objeto, permitindo assim que qualquer

[2]Na terminologia da área de gerenciamento de memória, a parte da aplicação que corresponde ao programa escrito pelo usuário é conhecida como *mutator*. Do ponto de vista do gerenciamento de memória, a única coisa que o mutator faz é mudar o estado da memória. Neste trabalho usamos o termo *programa cliente* em vez de mutator.

PUC-Rio - Certificação Digital Nº 0310845/CA

recurso computacional seja gerenciado seguindo uma política análoga à utilizada na reciclagem da memória heap. A sua origem está ligada à necessidade de linguagens de programação oferecerem uma expressividade adequada para representar tipos abstratos de dados: o uso de finalizadores ou *destrutores*, juntamente com construtores, pode garantir que certas propriedades sejam sempre preservadas ao longo do ciclo de vida de um objeto[3].

Existem dois mecanismos básicos de finalização associados a tipos abstratos de dados [35]. Na *finalização baseada em classes*, amplamente suportada em linguagens orientadas a objetos, todas as instâncias de uma classe tornam-se finalizáveis se esta classe implementar a interface padrão de finalização. Tal interface, geralmente descrita pela especificação da linguagem, define um método que é invocado automaticamente pelo coletor de lixo[4]. Na *finalização baseada em coleções* (containers), objetos tornam-se finalizáveis quando são inseridos em alguma estrutura de dados especial reconhecida pelo coletor de lixo. Neste caso a finalização não é inerente à classe ou ao tipo do objeto; finalizadores podem ser associados dinamicamente a instâncias específicas, o que permite que vários objetos, por vezes de tipos distintos, compartilhem um mesmo finalizador.

Referências fracas, também conhecidas como *ponteiros fracos* ou *referências virtuais*, são referências que não evitam que o objeto referenciado seja reclamado pelo coletor de lixo, como acontece com referências normais. Uma das motivações originais para o desenvolvimento e uso desta abstração foi a dificuldade associada à coleta de referências cíclicas em coletores de lixo que usam exclusivamente contagem de referências para rastrear objetos [20]. A utilização de referências fracas para implementar estruturas de dados potencialmente cíclicas evita vazamentos de memória neste tipo de sistema. Mais recentemente porém, com a ampla adoção de coletores de lixo baseados em rastreamento, o emprego de referências fracas passou a ser motivado sobretudo por constituir-se em uma opção elegante para que aplicações exerçam um nível maior de controle sobre a dinâmica de desalocação de memória.

[3] Algumas linguagens, como por exemplo C#, PHP, Perl e Python, utilizam o termo destrutor para o que definimos aqui como finalizador. Obviamente essa denominação acarreta bastante confusão entre programadores. Neste trabalho iremos reservar o termo destrutor apenas para denotar rotinas invocadas implicitamente antes da desalocação de um objeto em linguagens sem gerenciamento automático de memória. No Capítulo 3 discutimos o exemplo mais conhecido deste mecanismo: destrutores C++.

[4] Em algumas linguagens este método não pode ser invocado explicitamente.

PUC-Rio - Certificação Digital Nº 0310845/CA

1.1 Objetivos

Inúmeras linguagens de programação, ao menos desde meados da década de 80 [54, 70], oferecem suporte a alguma forma de referência fraca. Não obstante, de maneira geral este mecanismo permanece relativamente desconhecido entre programadores. Mesmo entre pesquisadores e desenvolvedores de linguagens existe muito pouco consenso quanto à sua semântica, que conforme iremos discutir mais adiante, varia consideravelmente entre diferentes implementações.

Isso contrasta com finalizadores, que contam com um suporte e interfaces mais homogêneos, sobretudo em linguagens orientadas a objetos, e são relativamente bem conhecidos por programadores (o uso de finalizadores porém permanece controverso, e constitui uma fonte frequente de erros). Além disso a semântica de finalizadores foi explorada por diversos autores (por exemplo, [58, 12, 34, 27, 18]), mesmo que apenas informalmente, enquanto a semântica de referências fracas, ao que sabemos, foi abordada de forma limitada em apenas dois artigos ([35, 42]), e mais recentemente em [48]. Tanto no que tange a finalizadores quanto a referências fracas podemos observar que inúmeras questões semânticas permanecem relativamente obscuras, dificultando a implementação, e limitando o suporte e o uso mais amplo destas facilidades.

Assim, o objetivo deste trabalho é explorar os conceitos de finalizadores e de referências fracas, suprindo a ausência de uma especificação clara e abrangente, e permitindo uma melhor compreensão, implementação e uso dos mecanismos correspondentes. Como ponto de partida realizamos um amplo levantamento sobre como é feito o suporte a finalizadores e referências fracas em diferentes linguagens de programação, buscando identificar as características comuns, os problemas, e as questões semânticas mais relevantes associadas a essas implementações. Concomitantemente, pesquisamos exemplos reais que ilustram as dificuldades típicas encontradas por programadores no uso destas facilidades.

Para garantir uma maior precisão durante essa análise, decidimos empregar um modelo abstrato de uma linguagem de programação com gerenciamento automático de memória. Considerando-se que os poucos modelos formais conhecidos que descrevem explicitamente o processo de coleta de lixo não são suficientemente expressivos, ou não podem ser facilmente estendidos para atender adequadamente aos requisitos deste trabalho, desenvolvemos um modelo próprio usando uma semântica operacional. Através deste modelo especificamos formalmente a semântica de finalizadores e referências fracas, incluindo descrições das

PUC-Rio - Certificação Digital Nº 0310845/CA

suas principais variantes e mecanismos relacionados.

A partir dessa análise formal exploramos inúmeras dificuldades inerentes à especificação e à implementação de tais facilidades. Em particular, provamos certas invariantes associadas a linguagens de programação com gerenciamento automático de memória, indicando como estas são afetadas pela introdução desses mecanismos.

Por fim, consideramos possíveis estratégias de implementação de finalizadores e referências fracas em diferentes tipos de sistemas. Algumas das opções semânticas investigadas impõem um custo de processamento expressivo, o que frequentemente inviabiliza a sua adoção na prática.

1.2 Exemplos de Uso

Finalizadores e referências fracas são abstrações relativamente pouco usadas pela maioria dos programadores. Contudo, conforme demonstra o amplo suporte que recebem das principais linguagens de programação com gerenciamento automático de memória, constituem-se em soluções elegantes para tratar um conjunto restrito mas relevante de problemas. Como motivação para este trabalho, descrevemos a seguir algumas destas aplicações.

1.2.1 Finalizadores

O uso de finalizadores é comum nos seguintes casos:

- **Desalocação de Memória** - Sistemas com gerenciamento automático de memória por vezes precisam interagir com rotinas desenvolvidas em linguagens que não contam com esse tipo de suporte. Finalizadores podem ser usados para liberar a memória alocada com uma rotina nativa como `malloc`, ampliando a região de memória gerenciada pelo coletor de lixo.

- **Desalocação de Recursos Externos** - Usualmente recursos externos, como descritores de arquivos e conexões, são representados em um programa através de um objeto proxy, cujo ciclo de vida segue fielmente o padrão de utilização do recurso correspondente. Por conseguinte, construtores e finalizadores podem ser usados para alocar e desalocar automaticamente o recurso associado ao proxy, estendendo a capacidade de gerenciamento do coletor de lixo, e evitando a implementação de rotinas adicionais.

PUC-Rio - Certificação Digital Nº 0310845/CA

Conforme iremos discutir no Capítulo 4, essa solução nem sempre é adequada em sistemas que empregam coletores de lixo baseados em rastreamento. O indeterminismo inerente a invocação de finalizadores pode atrasar demasiadamente a liberação dos recursos, afetando negativamente o desempenho da aplicação. Em alguns casos porém é possível atenuar esse problema forçando explicitamente a execução do coletor de lixo e dos finalizadores dos proxies coletáveis. Esses finalizadores devem ser invocados sempre que a disponibilidade do recurso externo cair abaixo de níveis pré-definidos.

- **Obtenção de Informações de Conectividade** - Boehm [18] descreve uma aplicação que usa grafos acíclicos direcionados (DAGs) contendo descritores de arquivos em seus nós terminais. Em função do tamanho e complexidade das estruturas de dados utilizadas, bem como do padrão de acesso da aplicação, é extremamente difícil rastrear explicitamente todas as referências para cada descritor de arquivos. A solução encontrada foi habilitar finalizadores para os nós terminais, fechando automaticamente os arquivos.

- **Coleta de Referências Cíclicas** - Conforme iremos discutir no Capítulo 2, coletores de lixo que utilizam exclusivamente contagem de referências são incapazes de reciclar a memória associada a estruturas de dados cíclicas. Christiansen e Torkington [24] apresentam um exemplo interessante que ilustra como finalizadores podem ser usados em Perl para eliminar automaticamente os ciclos associados a estruturas complexas, evitando o vazamento de memória. A solução proposta consiste na definição de um finalizador para qualquer tipo ou classe que represente uma estrutura de dados inerentemente ou potencialmente cíclica (anéis, grafos, listas duplamente encadeadas, etc). Quando a contagem de referências do objeto que contém a estrutura cíclica atinge zero, o finalizador é invocado, tendo como responsabilidade principal eliminar explicitamente os ciclos existentes na estrutura. Isso garante que todos os objetos componentes sejam coletados imediatamente após a última referência para a estrutura ser destruída.

- **Fallback** - Aplicações que fazem uso intensivo de recursos computacionais finitos, e não contam com o suporte de um coletor de lixo baseado em contagem de referências, geralmente devem realizar a desalocação explícita de tais recursos. Como erros nesse processo de desalocação são bastante comuns, finalizadores podem ser usados como um mecanismo de segurança (fallback) para liberar os recursos externos que deveriam ter sido liberados

PUC-Rio - Certificação Digital Nº 0310845/CA

explicitamente. Ainda que não existam garantias quando ou mesmo se um finalizador vai ser invocado, esta opção ainda é melhor do que a certeza de nunca ser invocado (o que acontece sempre que o programador esquece de liberar o recurso). Algumas classes da biblioteca padrão da linguagem Java utilizam essa solução [66].

- **Reciclagem de Objetos** - Uma aplicação pode obter ganhos de desempenho significativos através da reciclagem de objetos cujo custo de criação é relativamente alto. Uma forma de efetuar essa reciclagem é através da definição de finalizadores para todos os objetos recicláveis. Os finalizadores devem decidir se o objeto correspondente será coletado ou reciclado dependendo do número de objetos recicláveis disponíveis no sistema, da memória livre, ou de outros parâmetros relevantes. Novos objetos só são criados quando não existirem objetos recicláveis livres.

  Note que em linguagens que não permitem que o finalizador associado a um objeto seja executado mais de uma vez, como é o caso de Java, não é possível implementar esse tipo de solução.

PUC-Rio - Certificação Digital Nº 0310845/CA

1.2.2 Referências Fracas

O uso de referências fracas é comum nos seguintes casos:

- **Cache** - Aplicações que utilizam estruturas de dados de tamanho significativo podem melhorar sensivelmente o seu desempenho ao manter estas estruturas residentes na memória. Essa política contudo pode levar a um rápido esgotamento da memória livre. O uso de referências fracas possibilita a implementação de caches que preservam os objetos apenas enquanto o nível de utilização da memória não for restritivo.

  Considere como exemplo uma aplicação que utiliza uma tabela (um result set) com milhares de dados provenientes de um banco de dados. Após carregar os dados, a aplicação remove todas as referências para a tabela, exceto por uma referência fraca. Sempre que for necessário acessar algum dado na tabela, a aplicação primeiro verifica se a mesma ainda está acessível. Caso a tabela tenha sido coletada a aplicação recarrega os dados diretamente do banco de dados através de uma nova consulta.

  Uma outra aplicação interessante de caches são funções memorizadas (memoized functions) [40, 42]. O esforço computacional necessário para atender

a invocação de uma função complexa pode ser reduzido salvando-se o resultado de cada invocação. Quando a função é chamada novamente com o mesmo argumento, ela retorna o valor salvo. A utilização de um cache evita que a memorização dos resultados leve a um esgotamento da memória, preservando os resultados acessados mais recentemente (e possivelmente mais frequentemente).

Vale observar que em sistemas com coletores de lixo que usam contagem de referências os objetos são coletados imediatamente após a sua contagem atingir zero, o que impede a implementação de caches usando-se apenas o mecanismo básico de referências fracas.

- **Coleta de Referências Cíclicas** - O uso de referências fracas constitui uma solução alternativa para contornar o problema de coleta de referências cíclicas em sistemas que empregam coletores de lixo baseados em contagem de referências. Para evitar vazamentos de memória basta substituir referências ordinárias por referências fracas, de tal forma que qualquer ciclo de referências seja composto por ao menos uma referência fraca.

  Analogamente, referências fracas podem permitir a reciclagem de memória em uma situação menos comum: *ciclos externos*. Considere por exemplo uma linguagem de script que usa um componente de uma biblioteca de interfaces gráficas (GUI) desenvolvida na linguagem C. O script pode representar o componente como um proxy que mantém uma referência para o objeto C, o qual efetivamente implementa o componente. Se o componente suportar eventos através de um mecanismo de callback, ele eventualmente precisará manter uma referência para o proxy que irá executar a rotina de callback associada ao evento de interesse. Mas a referência do componente para o proxy não deveria impedir que o proxy fosse coletado, o que pode ser garantido se a referência para o proxy for fraca.

- **Tabelas de Propriedades** - Por razões diversas, algumas vezes é necessário associar atributos adicionais a um objeto sem modificar ou empregar a sua estrutura interna. Em aplicações orientadas a objetos por exemplo, pode ser necessário atribuir dinamicamente uma ou mais propriedades a um objeto de maneira independente da sua classe. Certamente podemos fazer tal associação utilizando uma estrutura de dados externa, como uma tabela associativa. No entanto, se o objeto em questão for usado como chave nessa tabela, a referência interna da tabela impedirá que o objeto seja coletado.

PUC-Rio - Certificação Digital Nº 0310845/CA

Para corrigir esse comportamento indesejado basta usar como chave de busca uma referência fraca para o objeto, ao invés do próprio objeto.

- **Conjuntos Fracos** - Conjuntos fracos (weak sets) constituem uma solução de implementação interessante sempre que objetos tem que ser processados como um grupo, mas sem interferir no ciclo de vida de cada um. Um exemplo bastante comum é o padrão de projeto *observador* (observer) que define "uma dependência de um-para-vários entre objetos de tal forma que quando o estado de um objeto se modifica, todos os seus objetos dependentes são notificados e tem o seu estado atualizado automaticamente" [30]. Essa notificação geralmente é feita pelo objeto observado, que portanto precisa manter referências para todos os seus observadores. O uso de referências fracas permite manter um acoplamento mínimo entre os objetos, o qual não impede que os observadores sejam coletados.

- **Finalização** - Referências fracas também podem ser utilizadas para informar o programa cliente que um objeto foi coletado, eventualmente disparando automaticamente rotinas associadas a tal evento. Essa dinâmica constitui um exemplo do mecanismo de finalização baseado em coleções, e evita alguns dos problemas associados a finalizadores baseados em classes, conforme iremos discutir no Capítulo 4.

PUC-Rio - Certificação Digital Nº 0310845/CA

1.3 Organização

Esta tese está estruturada da seguinte forma. Com o intuito básico de facilitar a compreensão do leitor, no Capítulo 2 apresentamos uma breve descrição das principais técnicas de coleta de lixo. No Capítulo 3 apresentamos um levantamento de como é o suporte oferecido pelas principais linguagens de programação com gerenciamento automático de memória a finalizadores e a referências fracas. No Capítulo 4 identificamos e discutimos de maneira informal as questões semânticas que acreditamos terem maior relevância sob o ponto de vista prático. Neste mesmo capítulo exploramos também alguns problemas típicos associados ao uso e à implementação de finalizadores e referências fracas. No Capítulo 5 desenvolvemos um modelo abstrato através do qual especificamos uma semântica formal para coleta de lixo, finalizadores, referências fracas e para os principais mecanismos associados a estas abstrações. Provamos ainda várias invariantes inerentes a

linguagens de programação que contam com tais facilidades. No Capítulo 6 investigamos alternativas de implementação, e por fim, no Capítulo 7 apresentamos nossas conclusões.

PUC-Rio - Certificação Digital Nº 0310845/CA