1
Introdução

Muitas linguagens de programação fornecem coleta de lixo a fim de desalocar automaticamente objetos inacessíveis. Os coletores de lixo livram o programador da responsabilidade de desalocar a memória, simplificando assim a implementação de algoritmos complexos e eliminando erros por falta de memória que são difíceis de detectar e corrigir. Ainda hoje, o gerenciamento dinâmico e automático de memória é uma área bastante ativa dentro da Ciência da Computação (Leal05). Essa constante busca por melhorias nos coletores de lixo se deve não só à necessidade de administrar a memória de forma mais eficiente, mas também à demanda de um desenvolvimento mais acelerado de sistemas de computação tradicionais.

Duas das ferramentas mais poderosas da engenharia de software são abstração e modularidade. O gerenciamento explícito de memória vai fortemente contra esses princípios (Jones96). Além disso, o trabalho realizado por Rovner (Rovner85) sugere que uma porção considerável do tempo de desenvolvimento pode ser gasta para resolver problemas relacionados ao gerenciamento explícito de memória, sobretudo à correção de erros. Ele estimou que 40% do tempo despendido no desenvolvimento do sistema Mesa foi dedicado à solução desses problemas. No entanto, coletores de lixo fornecem ao programador um aumento na abstração, ou seja, o programador não deve se preocupar com detalhes de gerenciamento de memória e podem focar sua atenção na implementação da real funcionalidade da aplicação. Além disso, coletores permitem uma programação mais modular, já que um módulo não precisa saber se um objeto está sendo referenciado por outros módulos para que possa liberar, ou não, a memória ocupada por ele explicitamente.

Coletores de lixo realizam o trabalho de gerenciamento automático de memória nos bastidores, ou seja, o programa não toma conhecimento sobre as ações executadas pelo coletor. Dizemos então que o coletor de lixo age de forma transparente. No entanto, em várias situações as aplicações requerem um maior grau de flexibilidade, que não é possível em coletores totalmente transparentes. Nesse caso, é interessante permitir que o programa escrito pelo usuário interaja diretamente com o coletor de lixo, quer seja alterando o

seu comportamento, fornecendo a ele informações extras ou ainda obtendo informações geradas a partir de sua execução. Sendo assim, muitas linguagens fornecem funções e mecanismos auxiliares a fim de obter essa interação, ou seja, elas fornecem uma interface com o coletor de lixo. Tais interfaces são tipicamente representadas por *finalizadores* (Atkins88, Boehm03, Dybvig93, Hayes92) e *referências fracas* (Leal05).

Finalizadores são rotinas executadas automaticamente pelo coletor de lixo antes de liberar a memória ocupada por um objeto. Eles vêm sendo utilizados em várias atividades, incluindo no gerenciamento de caches de objetos e na liberação de recursos providos por servidores ou outros programas. Veremos na Seção 2.1.2 que finalizadores possuem uma natureza assíncrona. Devido a isso, Bloch (Bloch01) chega a dizer em seu trabalho que finalizadores são imprevisíveis, frequentemente perigosos e desnecessários e afetam negativamente o desempenho do programa. No entanto, Boehm (Boehm03) argumenta que o uso de finalizadores é essencial em alguns casos e seu assincronismo não deve necessariamente levar a programas não confiáveis.

Referências fracas são um tipo especial de referência que não impede que um objeto seja coletado pelo coletor de lixo. Muitas implementações de linguagens que utilizam coleta de lixo, ao menos desde a década de 80, apresentam algum suporte a referências fracas (Xerox85, Rees84). Dentre outras aplicações, elas podem ser empregadas como um mecanismo de finalização, mas evitando diversas dificuldades associadas a finalizadores tradicionais. Dependendo da linguagem de programação e da forma como é feito o controle da coleta de lixo, o suporte a finalizadores torna-se completamente desnecessário.

O trabalho apresentado por Leal (Leal05) procurou explorar os conceitos de finalizadores e referências fracas, suprindo a ausência de uma especificação clara e abrangente. Tomando como ponto de partida esse trabalho, apresentamos um estudo detalhado sobre como implementar mecanismos de finalização através de referências fracas. Com isso, podemos substituir finalizadores tradicionais por referências fracas e assim simplificar a implementação das linguagens de programação.

Com essa proposta de implementar finalizadores via referências fracas, estamos sugerindo um uso maior desse último mecanismo. Isso se deve ao fato de que iremos usar referências fracas não só nos usos típicos conhecidos desse mecanismo (apresentados no Capítulo 3), como também iremos usar referências fracas em cada um dos usos típicos de finalizadores encontrados. Dessa forma, estamos explorando mais as implementações de referências fracas presentes nas linguagens de programação. Contudo, a maioria das implementações de referências fracas apresenta o problema de ciclos em tabelas fracas.

Em muitos casos, queremos adicionar dinâmicamente propriedades a um objeto independente dos atributos de sua classe. Para isso, uma opção bastante comum é utilizar uma *tabela de propriedades* onde uma referência para o objeto é inserida como chave de busca e o respectivo valor armazena as propriedades extras. No entanto, se todas as referências na tabela de propriedades forem comuns, o simples fato inserir um novo par chave/valor garante que o objeto referenciado pela chave nunca será coletado. Para resolver esse problema o ideal é utilizar uma estrutura de dados chamada *tabelas fracas*, implementada através de referências fracas. Essa estrutura é constituída de *pares fracos* (weak pairs) onde o primeiro elemento do par, a chave, é mantido por uma referência fraca e o segundo elemento, o valor, é mantido por uma referência comum. Dessa forma, a adição de propriedades a um objeto não irá modificar o instante em que ele deve ser coletado

Porém, um grande problema com tabelas fracas ainda persiste na maioria das linguagens. A existência de referências cíclicas entre chaves e valores impede que os elementos que compõem o ciclo sejam coletados, mesmo que eles não sejam mais utilizados pelo programa. Isso acaba ocasionando um grande desperdício de memória, impossibilitando o uso de tabelas fracas. As linguagens Java (SUN04) e Lua (Ierusalimschy06), por exemplo, apresentam esse problema e, através da lista de discussão da linguagem Lua, pudemos observar que é motivo de reclamações constantes por parte dos programadores. Uma solução para o problema de ciclos em tabelas fracas foi encontrada pela linguagem Haskell (Haskell07). A implementação do mecanismo de referências fracas dessa linguagem possui uma adaptação do mecanismo de *ephemerons* apresentado por Hayes (Hayes97). Baseados no sucesso obtido em Haskell, realizamos uma adaptação do mesmo mecanismo para Lua, resolvendo o problema nessa linguagem.

1.1 Objetivos

As contribuições desta pesquisa se situam na área de Linguagens de Programação, mais especificamente no desenvolvimento de sistemas de coleta de lixo que fornecem suporte a finalizadores e referências fracas a fim de prover uma maior flexibilidade às aplicações. Inúmeras linguagens de programação fornecem algum tipo de suporte a ambos os mecanismos. Contudo, é possível utilizar referências fracas como um mecanismo alternativo de finalização, tornando desnecessário, em determinados contextos, o suporte a finalizadores e simplificando a linguagem.

Assim, o um dos objetivos deste trabalho é substituir finalizadores

tradicionais por um mecanismo de finalização baseado em referências fracas, seja via notificação passiva ou ativa. Realizamos um estudo detalhado de quais extensões são necessárias ao mecanismo de referências fracas a fim de dar suporte a finalização em diferentes contextos (linguagem de programação, modo de controle da coleta de lixo, etc). Com isso, analisamos as vantagens e desvantagens de utilizar referências fracas no lugar de finalizadores. Discutimos também a possibilidade de utilizar apenas notificação passiva e quais as vantagens desse mecanismo se comparado a notificação ativa. Além disso, implementamos um mecanismo de finalização baseado em referências fracas para a linguagem Lua (Ierusalimschy06).

Outro objetivo deste trabalho diz respeito ao problema de ciclos em tabelas fracas. Como ponto de partida, estudamos detalhadamente o funcionamento do mecanismo de ephemerons. Em seguida, estudamos o funcionamento do coletor de lixo de Lua a fim de estabelecer a melhor adaptação do mecanismo de ephemerons a ser implementada. Posteriormente, incorporamos esse mecanismo ao algoritmo de coleta de lixo de Lua, resolvendo assim o problema de ciclos nessa linguagem.

1.2 Organização

Esta dissertação está organizada da seguinte forma. Inicialmente, no Capítulo 2, descrevemos os conceitos necessários para um melhor entendimento do nosso trabalho. Com o intuito de identificar os problemas abordados por finalizadores e referências fracas, apresentamos no Capítulo 3 uma descrição dos principais usos desses mecanismos. Nesse capítulo, para cada uso típico de finalizadores encontrado, descrevemos uma solução baseada em referências fracas e discutimos a notificação passiva como a melhor alternativa em vários casos. No Capítulo 4, descrevemos um mecanismo de finalização baseado em referências fracas para a linguagem Lua. Também apresentamos nesse capítulo algumas linguagens de programação onde o mecanismo de finalização é implementado através de referências fracas e discutimos cada implementação. No Capítulo 5, mostramos como ephemerons foram adaptados à implementação do coletor de lixo de Lua para resolver o problema de ciclos em tabelas fracas. Por fim, no Capítulo 6, apresentamos as conclusões do nosso trabalho.