4
Mecanismos de Finalização Baseados em Referências Fracas

Este capítulo descreve um mecanismo de finalização baseado em referências fracas para a linguagem Lua. Além disso, apresentamos algumas linguagem de programação cujo mecanismo de referências fracas pode ser utilizado na construção de finalizadores. As Seções 4.1, 4.2 e 4.3, discutem respectivamente os mecanismo de finalização de Modula-3, Python e Haskell. Nosso objetivo é mostrar que em algunas linguagens o suporte a finalizadores já pode ser considerado desnecessário, pois referências fracas podem ser usadas para implementar a finalização de objetos. No entanto, a linguagem Python mostrada aqui ainda oferece finalizadores tradicionais. Ao contrário do que consideramos ideal para a implementação de finalizadores, todas as linguagens descritas utilizam um mecanismo de notificação ativa para implementar as rotinas de finalização.

Na Seção 4.4, descrevemos nossa implementação de um mecanismo de finalização baseado em referências fracas para a linguagem Lua. Esse mecanismo foi adaptado à implementação atual do coletor de lixo de Lua. Ao contrário dos outros mecanismos descritos, optamos por utilizar a notificação passiva, pois evita os problemas de concorrência e sincronização tratados na Seção 3.2.

4.1
Modula-3

Modula-3 segue o paradigma imperativo e possui suporte para tipos parametrizados, multithreading, tratamento de exceções e coleta de lixo. A fim de fornecer um maior controle sobre a coleta de lixo, Modula-3 possui um mecanismo de referências fracas que pode ser utilizado através da interface `WeakRef`.

Referências fracas são criadas através do método `FromRef`. Esse método recebe como parâmetro uma referência ordinária para um objeto qualquer alocado na memória heap e retorna uma referência fraca para esse objeto. Uma referência fraca é removida quando o coletor de lixo determina que o objeto não é mais alcançável pelo programa. A definição de um objeto alcançável

para Modula-3 será dada mais adiante. Uma vez que uma referência fraca é removida, ela não pode ser ressuscitada, mesmo que o objeto ao qual ela se refere se torne alcançável novamente. Um callback pode ser associado a uma referência fraca através de um parâmetro opcional do método `FromRef`. Caso o callback não seja nulo, o coletor de lixo irá executá-lo quando determinar que o objeto referenciado tornou-se inacessível, e após remover a referência fraca. O callback sempre é executado antes do objeto associado ser efetivamente desalocado e recebe como parâmetro uma referência ordinária para o próprio objeto, podendo então ressuscitá-lo. Mais de um callback pode ser associado a uma referência fraca, mas a especificação da linguagem não oferece qualquer garantia quanto à ordem de execução (Modula07).

De acordo com a definição da linguagem, um objeto é alcançável se existe um caminho de referências para ele a partir de variáveis globais, valores nos registros de ativação ativos, ou ainda a partir de um nó fracamente referenciado que possua um callback não nulo. Sendo assim, uma referência fraca para um objeto A não o torna alcançável, porém, se ele possui um callback, os objetos referenciados por A serão alcançáveis. Dessa forma, agregações de objetos são sempre finalizadas seguindo uma ordem correta, ou seja, seguindo o sentido das referências entre os objetos. Por outro lado, referências cíclicas impedem a finalização (e a coleta) de objetos finalizáveis.

A Listagem 5 foi retirada do trabalho de Leal (Leal05) e ilustra a implementação de um rotina de finalização em Modula-3 (o símbolo "..." foi colocado onde o código não é relevante para o exemplo). Em Modula-3, `WeakRef.T` é uma estrutura de dados que referencia um objeto sem impedir que ele seja coletado, ou seja, representa uma referência fraca. Além disso, `REFANY` é um tipo da linguagem que representa todas as referências que podem ser rastreadas. Voltando ao exemplo, temos que `MyStream` representa uma classe qualquer que envia dados através de um stream e utiliza um buffer. Ao tornar-se inacessível, um objeto dessa classe deve esvaziar o buffer e fechar o stream correspondente. O método `New` cria um novo stream que é armazenado na variável `res` e, a seguir, cria uma referência fraca para o novo stream através do método `FromRef`. Nesse exemplo, como a referência fraca não é utilizada o resultado da função é ignorado, o que justifica o uso de `EVAL`. Porém, o coletor continua tendo acesso a essa referência, pois todas as referências fracas criadas por um programa são armazenadas pelo coletor numa estrutura interna. O método `Cleanup` implementa a rotina de finalização. O parâmetro `self` desse método representa a referência fraca que tem `Cleanup` como rotina de finalização. O parâmetro `ref` representa uma referência ordinária para o objeto a ser finalizado. A implementação de `FromRef` se encarrega de associar

a referência fraca criada ao parâmetro `self` e o objeto ao parâmetro `ref`.

Listagem 5 Implementação de uma rotina de finalização em Modula-3

```
MODULE MyStream; IMPORT WeakRef, Wr, ...;

  PROCEDURE New(...): Stream =
    (* Inicializa res como sendo um Stream *)
    VAR res := NEW (Stream);
    BEGIN
      ...
      (* Cria uma referência fraca e associa um callback a ela *)
      EVAL WeakRef.FromRef(res, Cleanup);
      RETURN res
  END New;
  ...

  (* Callback representando uma rotina de finalização *)
  PROCEDURE Cleanup(READONLY self: WeakRef.T; ref: REFANY) =
    VAR wr: Stream := ref;
    BEGIN
      IF NOT Wr.Closed(wr) THEN
        (* esvazia o buffer e fecha o stream *)
        Wr.Flush(wr);
        Wr.Close(wr);
      END
  END Cleanup;

END MyStream.
```


4.2 Python

O módulo `weakref` implementa o mecanismo de referências fracas de Python. Para criar uma referência fraca, basta utilizar a função `ref` passando o objeto a ser referenciado como parâmetro. Essa função retorna um objeto que representa uma referência fraca para o objeto original. A função `ref` também aceita um segundo parâmetro opcional, um callback que é invocado quando o objeto referenciado torna-se inalcançável (mas antes de ser coletado, ou mesmo finalizado). O objeto que representa a referência fraca será passado como único parâmetro do callback. Se mais de um callback estiver associado a um objeto, os callbacks serão executados na ordem inversa em que foram registrados.

Além de oferecer um mecanismo de finalização baseado em referências fracas, finalizadores em Python também podem ser implementados através do método `__del__`. Dessa forma, o finalizador é acoplado à classe, seguindo a

semântica típica de lingagens orientadas a objeto. Em Python, é possível implementar tanto finalização baseada em coleções, utilizando callbacks, quanto finalização baseada em classes, através do método `__del__`. A principal diferença é que callbacks podem ser associados dinamicamente às instâncias de uma classe. Além disso, pode-se associar múltiplos callbacks a um objeto.

4.3 Haskell

O Glasgow Haskell Compiler (GHC) implementa referências fracas através de pares chave/valor, onde o valor é considerado acessível se a chave for acessível, mas a conectividade do valor não influencia na conectividade da chave. Durante a coleta de lixo, o campo referente ao valor de uma referência fraca não é rastreado a não ser que a chave seja alcançável. Esse tipo de referência fraca é uma generalização das referências fracas comuns usadas na criação de mapeamentos fracos com dinâmicas de coleta complexas, como *memo tables* (Jones00).

O GHC permite que sejam associadas ações[1] (basicamente callbacks, que em GHC são denominados finalizadores) a referências fracas, as quais são executadas após as chaves serem limpas. Se múltiplos finalizadores forem associados à mesma chave, estas serão executadas em uma ordem arbitrária, ou mesmo de forma concorrente. O modo mais simples de criar uma referência fraca é através da função `mkWeakPtr` que recebe um valor de um tipo qualquer e um finalizador do tipo `IO()` e retorna uma referência fraca do tipo `Weak a` referente ao valor.

A documentação do GHC (GHC07) especifica a semântica de referências fracas baseadas nos pares chave/valor. Um objeto é coletável se: (a) ele é referenciado diretamente por um objeto alcançável, ao invés de uma referência fraca, (b) é uma referência fraca cuja chave é alcançável ou (c) é o valor ou finalizador de uma referência fraca cuja chave é alcançável. Note que um ponteiro do valor ou do finalizador para a chave associada não torna a chave alcançável. No entanto, se a chave puder ser alcançada de outra forma, então o valor e o finalizador são alcançáveis e consequentemente qualquer outra chave que seja referenciada por eles direta ou indiretamente. O GHC garante ainda que finalizadores registrados serão executados uma única vez, quer seja quando a chave correspondente for limpa, ou através de uma invocação explícita, ou ainda ao final da execução da aplicação. Essa especificação de referências fracas é semelhante à semântica de ephemerons, diferindo em alguns detalhes,

[1] Em linguagens puramente funcionais, efeitos colaterais e estados globais podem ser representados através de *monads*. Mais detalhes sobre esse conceito podem ser encontrados em nos trabalhos de Wandler (Wandler95, Wandler90).

como no tratamento de finalizadores. Uma comparação mais detalhada sobre o mecanismo de ephemerons e a implementação de referências fracas de Haskell pode ser encontrado no trabalho de Jones (Jones00).

4.4 Mecanismo de Notificação Passiva para Lua

A implementação atual da linguagem Lua oferece suporte a um mecanismo de finalização que pode ser usado exclusivamente com um tipo específico, `userdata`. Para tornar um `userdata` finalizável, deve-se registrar um finalizador para esse objeto (através da definição do metamétodo `__gc`). Após o coletor de lixo determinar que um objeto não pode ser mais acessado pelo programa, ele insere o finalizador correspondente a esse objeto em uma fila. Ao final do ciclo de coleta de lixo, os finalizadores são invocados, recebendo como parâmetro o próprio objeto. Lua garante que todos os finalizadores serão invocados antes do término da aplicação.

Vimos na Seção 2.1.1 que o coletor de lixo de Lua é baseado na técnica de rastreamento. Sendo assim, não é possível determinar quando os finalizadores serão executados. No Capítulo 3, vimos que esse indeterminismo pode afetar negativamente o desempenho da aplicação. O mesmo ocorre caso o mecanismo de finalização seja baseado em notificação ativa, usando callbacks. Nesse caso, o coletor de lixo também irá decidir quando executar os callbacks associados às referências fracas, e não é possível determinar quando isso irá ocorrer. Além disso, vimos que a execução de callbacks e finalizadores pode introduzir linhas de execução concorrentes na aplicação e, caso a linguagem utilizada não ofereça um suporte adequado à concorrência e sincronização, o uso desses mecanismos pode levar a inconsistências durante a execução do programa.

A fim de fornecer ao programador um maior controle sobre a coleta através de um mecanismo de finalização simples, decidimos incorporar à implementação atual do coletor de lixo de Lua um mecanismo de notificação passiva. Esse mecanismo, ao contrário dos finalizadores existentes na implementação atual da linguagem, pode ser usado com qualquer tipo de Lua, e não apenas com `userdata`. Mesmo perdendo um pouco da automação, pois o programador é responsável por acessar a fila de notificações, acreditamos que esse mecanismo de notificação passiva pode trazer mais vantagens que o mecanismo de finalizadores atual. Isso é justificado pelo fato de que a notificação passiva elimina o problema de concorrência e sincronização e atenua o problema do indeterminismo. Com esse mecanismo, o programa pode esperar por condições específicas para executar as ações associadas à finalização de um objeto. Além disso, o uso de notificação passiva elimina o problema de objetos ressucitáveis,

pois o objeto a ser finalizado continua acessível para o programa, ao contrário do que o ocorre com finalizadores.

A linguagem Lua implementa referências fracas através de sua principal estrutura de dados, os arrays associativos também conhecidos como *tabelas Lua*. Na Seção 2.3, vimos que, modificando o campo `__mode` de uma metatabela, o programador pode criar tabelas onde apenas a chave é fraca, apenas o valor é fraco, ou ambos são fracos. Essas tabelas são chamadas de tabelas fracas. O mecanismo de notificação passiva que implementamos está aclopado a essas tabelas de forma que a cada tabela fraca pode ser associada uma *tabela de notificações*[2]. Cabe ao programador estabelecer essa associação.

O programador pode optar por usar ou não uma tabela de notificações para uma determinada tabela fraca. Para isso, estabelecemos um novo campo para as metatabelas de Lua, o campo `__notify`. Caso o programador queira habilitar o uso de uma tabela de notificações para uma tabela fraca, ele deve atribuir a esse campo uma tabela vazia. Dessa forma, sempre que o coletor de lixo remover uma entrada da tabela fraca, ele irá copiar essa entrada para a tabela de notificações da metatabela, ou seja, para a tabela atribuída ao campo `__notify`. Caso o campo `__notify` seja nulo, ou caso o programador tenha atribuído a esse campo outro valor que não seja uma tabela, o coletor irá simplesmente remover a entrada da tabela fraca.

A Listagem 6 exemplifica de maneira simples o uso de uma tabela de notificações. Inicialmente, criamos uma tabela com chaves e valores fracos, atribuindo a string `"kv"` ao campo `__mode` da metatabela. Vale notar que uma tabela de notificações pode ser associada a qualquer tabela fraca, tenha ela apenas chaves fracas, apenas valores fracos, ou tanto chaves quanto valores fracos. Voltando ao exemplo, após criar a tabela fraca, atribuímos uma nova tabela ao campo `__notify` da metatabela. Em seguida, duas entradas são inseridas na tabela fraca. A referência do programa para a chave da primeira entrada é perdida quando atribuímos uma nova tabela a variável `key`. Em seguida, forçamos uma coleta chamando a função `collectgarbage` a fim de coletar a primeira entrada. Como definimos uma tabela de notificações para a tabela fraca, o coletor irá copiar a entrada para a tabela de notificações após removê-la da tabela fraca. O primeiro loop `for` irá imprimir a string `"2"`, pois apenas a segunda entrada permanece na tabela. O segundo loop, por sua vez, irá imprimir `"1"`, pois a primeira entrada foi movida para a tabela de notificações.

Diferentes tabelas fracas podem possuir uma mesma tabela de noti-

[2]Como em Lua a fila de notificações é representada através de uma tabela, iremos nos referir a ela como tabela de notificações.

Listagem 6 Criando uma tabela de notificações para uma tabela fraca.

```
wt = {} -- tabela fraca
mt = {} -- metatabela
setmetatable(wt, mt)
mt.__mode = "kv" -- Define os valores e as chaves como fracos
mt.__notify = {} -- Cria a tabela de notificacao

key={}
wt[key] = 1
key = {}
wt[key] = 2

collectgarbage()

for k, v in pairs(wt) do
print(v)
end
for k, v in pairs(mt.__notify) do
print(v)
end
```


ficações. Isso pode ocorrer de duas maneiras. A primeira delas é quando uma mesma metatabela é associada a diferentes tabelas fracas, e o valor do campo `__notify` da metatabela é uma tabela. Um exemplo desse caso é mostrado na Listagem 7, onde as tabelas fracas `a` e `b` foram associadas à metatabela `mt`. Duas entradas foram adicionadas à tabela `a` e à tabela `b`. Como as chaves são fracas e não existem referências para elas fora das tabelas `a` e `b`, todas as chaves serão coletadas e copiadas para a tabela de notificações. Como `a` e `b` possuem a mesma tabela de notificações, o loop `for` irá imprimir as strings `"1"`, `"2"`, `"3"` e `"4"`.

O segundo modo de construir uma mesma tabela de notificações para tabelas fracas diferentes é atribuindo uma mesma tabela a campos `__notify` de diferentes metatabelas, que por sua vez estão associadas às tabelas fracas. Um exemplo é mostrado na Listagem 8, onde a tabela `n` foi atribuída à tabela de notificações da tabela fraca `a` e à tabela de notificações da tabela fraca `b`. Nesse caso, todas as entradas removidas das tabelas fracas `a` e `b` serão copiadas para suas respectivas tabelas de notificações. Como essas tabelas referenciam a mesma tabela, a tabela de notificações de `a` irá conter as entradas removidas da tabela `b` e vice-versa, ou seja, elas serão a mesma tabela.

Existe um outro caso especial que devemos levar em consideração: quando um mesmo objeto é inserido em diferentes tabelas fracas. Digamos que duas tabelas fracas, A e B, possuam o objeto k como chave (o valor pode também

Listagem 7 Associando duas tabelas fracas a uma mesma metatabela.

```
a = {} -- tabela fraca
b = {}
mt = {} -- metatabela
setmetatable(a, mt)
setmetatable(b, mt)
mt.__mode = "k" -- Define os valores e as chaves como fracos
mt.__notify = {} -- Cria a tabela de notificacao

a[{}] = 1
a[{}] = 2
b[{}] = 3
b[{}] = 4
collectgarbage()

for k, v in pairs(mt.__notify) do
print(v)
end
```


referenciar um mesmo objeto ou objetos diferentes). Suponha que as chaves de A e B são mantidas por referências fracas e que não existem referências ordinárias para o objeto k, ou seja, k pode ser coletado. Sendo assim, caso as tabelas A e B possuam diferentes tabelas de notificações, k será copiado duas vezes: uma vez ao ser inserido na tabela de notificações de B e uma vez ao ser inserido na tabela de notificações de A. Contudo, caso as tabelas A e B possuam a mesma tabela de notificações, o coletor irá copiar k para essa tabela de notificações quando removê-lo da primeira tabela e, em seguida, irá copiar uma segunda vez ao remover da segunda, sobrepondo a primeira cópia. Consequentemente, apenas uma cópia de k pode ser encontrada na tabela de notificações.

Vamos mostrar agora como implementar finalização através do nosso mecanismo de notificação passiva. De modo geral, a tabela de notificações contém todos os objetos que podem ser finalizados. Sendo assim, o programador deve acessar essa tabela e executar para cada objeto encontrado, a rotina de finalização adequada. Quando a tabela de notificações não for mais útil ao programa, este também deve explicitamente limpar essa tabela, por exemplo, atribuindo `nil`, para que o coletor de lixo seja capaz de reciclar a memória associada a ela.

A Listagem 9 apresenta um uso real em Lua que utiliza uma implementação em C para arrays. Nesse caso, a variável `path` é uma String contendo o caminho para a biblioteca em C. As funções `new` e `clean` são respectivamente

Listagem 8 As tabelas de notificações de `a` e `b` referenciam a mesma tabela.

```
a = {}
b = {}
mta = {} -- metatabela de a
mtb = {} -- metatabela de b
setmetatable(a, mta)
setmetatable(b, mtb)
mta.__mode = "k"
mtb.__mode = "k"
n = {}
mta.__notify = n
mtb.__notify = n
```

funções da biblioteca para criar um novo array e limpar a memória ocupada pelo programa em C. Após criar um novo array e atribuí-lo à variável `a`, criamos uma tabela fraca com valores fracos e inserimos `a` como valor dessa tabela. Dessa forma, a tabela `wt` não impedirá que `a` seja coletado. Em seguida, criamos uma tabela de notificações para a tabela fraca `wt` e atribuímos `nil` à variável `a` para que o userdata antes armazenado seja coletado. Logo após, forçamos uma coleta de lixo. Por fim, o programa acessa explicitamente a tabela de notificações e, para cada objeto encontrado, executa a rotina `array.clean`, passando como parâmetro o próprio objeto.

A implementação atual desse mecanismo de notificação possui um problema crítico. Para entender esse problema, precisamos discutir alguns detalhes do funcionamento do coletor de lixo de Lua. Vimos, na Seção 2.1.1 que o coletor de Lua possui quatro fases: uma para rastrear os objetos, uma fase atômica, onde um conjunto de operações é executado em um único passo, uma fase para desalocar a memória ocupada pelos objetos não marcados e uma fase para invocar os finalizadores. Durante a fase de rastreamento, todas as tabelas fracas encontradas são colocadas em uma lista. Ao final dessa fase, e durante a fase atômica, o coletor percorre a lista de tabelas fracas e remove todos os pares que possuem chaves ou valores fracos que não foram alcançados pelo rastreamento, ou seja, não são referenciados fora da tabela fraca. Isso ocorre em uma função chamada `cleartable`, que não pode ser intercalada com a execução do programa e por isso está inserida na fase atômica do coletor. Em seguida, o coletor prossegue para as fases de desalocação de memória e execução dos finalizadores.

Neste capítulo, vimos que a tabela de notificações deve ser preenchida enquanto o coletor remove as entradas da tabela fraca. Sendo assim, adaptamos a função `cleartable` para que esse comportamento fosse obtido, ou seja,

Listagem 9 Exemplo de uso de notificação passiva para finalização de objetos.

```
array_Init, err1 = package.loadlib(path, "_luaopen_array")

if not array_Init then -- se a biblioteca não pode ser aberta
error(err1)
end

array_Init()

a = array.new(1000000)
wt = {a}
mt = {}
setmetatable(wt, mt)
mt.__mode = "v"
mt.__notify = {}

a = nil

collectgarbage()

for k, v in pairs(mt.notify) do
array.clean(v)
end
```


dentro dessa função preenchemos a tabela de notificações. Isso significa que durante a fase atômica, na execução de `cleartable`, o coletor precisa alocar memória para construir a tabela. O coletor de Lua executa um controle rígido sempre que precisa alocar memória. Esse controle abrange toda a implementação da coleta de lixo e é extremamente necessário para garantir que erros de memória não irão ocorrer durante a coleta. Contudo, na nossa implementação, não foi possível executar esse controle, pois toda a construção da tabela de notificações está contida dentro de `cleartable` que por sua vez é executada em um único passo. Caso não haja memória suficiente para executar essa construção, a própria coleta pode gerar um erro por falta de memória. Para impedir esse erro, teríamos que trabalhar de forma mais abrangente no coletor de Lua, não apenas modificando a função `cleartable`, mas sim realizando adaptações em todo o algoritmo. Esse trabalho é extremamente complexo e não foi possível elaborar uma solução. Consequentemente, não é aconselhável utilizar nossa implementação corrente do mecanismo de notificação passiva numa futura versão da linguagem Lua.