# 3
# Alternativas de Interface e Suporte

Neste capítulo descrevemos como algumas das mais conhecidas linguagens de programação com gerenciamento automático de memória dão suporte a finalizadores e referências fracas. Como muitas destas linguagens oferecem interfaces bem parecidas, iremos nos ater apenas às interfaces que julgamos ser mais relevantes, originais, ou que melhor ilustram uma semântica específica dentre as diferentes soluções que conseguimos identificar.

Para realizar essa avaliação consideramos um conjunto amplo de linguagens imperativas e funcionais, que incluiu C# [56, 7], C++ [61], Eiffel [48], Glasgow Haskell [6], Java [62], Lisp [31], Lua [40], Standard ML [5], Modula-3 [36, 2], OCaml [45], Perl [4], PHP [11], Python [65], Ruby [63], GNU Scheme [32] e Smalltalk [21, 8].

Uma análise semântica mais profunda das diferentes soluções descritas neste capítulo será postergada até o Capítulo 4.

## 3.1
## Finalizadores

Nesta seção descrevemos as principais interfaces associadas a finalizadores. Antes de explorarmos estas diferentes alternativas, cabe aqui uma breve discussão sobre uma facilidade bastante comum em linguagens imperativas: o bloco `try-finally`.

### 3.1.1
### try-finally

O bloco `try-finally` é utilizado basicamente para o tratamento de exceções [59]. Após abandonar o escopo definido pelo bloco `try`, mesmo que existam exceções pendentes (que não foram tratadas), o fluxo de execução do programa é imediatamente transferido para o bloco `finally`.

Essa facilidade pode ser vista como um mecanismo de finalização baseado na

estrutura sintática do programa. Considere o seguinte exemplo:

```
sem_t mutex; // declara um lock (mutex) no escopo global
...
{
  sem_wait(&mutex); // adquire o lock
  foo();            // invoca um método que pode lançar uma exceção
  sem_post(&mutex); // libera o lock
}
```

Qualquer exceção lançada por *foo* interrompe prematuramente a execução do bloco acima, fazendo com que o lock fique bloqueado indefinidamente. Usando um bloco `try-finally` esse problema é facilmente contornado.

```
sem_t mutex; // declara um lock (mutex) no escopo global
...
try {
  sem_wait(&mutex); // adquire o lock
  foo();            // invoca um método que pode lançar uma exceção
} finally {
  sem_post(&mutex); // libera o lock
}
```

Essa estrutura pode ser adaptada para efetuar a finalização de objetos. Por exemplo

```
Object obj;
try {
  obj = new Object();
  foo(obj);
} finally {
  // finaliza o objeto
  obj.releaseResouces();
}
```

Infelizmente esse padrão de finalização apresenta limitações óbvias, tanto do ponto de vista sintático quanto semântico, as quais dificultam ou mesmo inviabilizam a sua utilização irrestrita em programas de maior porte. Ademais, a necessidade de finalização quase sempre está associada a tipos abstratos de dados, e não à estrutura sintática do código [58, 34]. É mais simples fechar um arquivo

ou conexão através do finalizador de um proxy do que inserir blocos `try-finally` em cada região do programa em que este tipo de objeto é criado. Mecanismos de finalização atrelados a objetos são importantes não apenas por prover uma maior expressividade para representar tipos abstratos de dados, mas também como uma forma de simplificar a tarefa do programador.

### 3.1.2 C++

As principais implementações de C++, com exceção de versões desenvolvidas pela Microsoft para o ambiente .Net (Visual Studio .Net), não possuem gerenciamento automático de memória, e portanto não oferecem suporte a finalizadores tal como definimos aqui. Porém esta linguagem permite definir rotinas que são implicitamente executadas sempre que a memória associada a um objeto é desalocada. Tais rotinas, conhecidas como destrutores, são definidas como métodos associados a classes. Sua invocação acontece de forma síncrona quando o registro de ativação no qual o objeto foi alocado é destruído, ou quando o objeto é explicitamente desalocado através do comando `delete`. Essa dinâmica garante a finalização de objetos mesmo quando ocorrem exceções, e facilita a implementação de hierarquias de classes. Para instâncias de uma classe definida usando-se o mecanismo de herança, todos os destrutores que compõem a hierarquia da classe são implicitamente invocados, seguindo-se a ordem inversa de declaração das classes. Para objetos declarados em um bloco de código, a ordem de execução dos destrutores segue a ordem inversa de declaração dos objetos.

Com o objetivo principal de evitar os problemas inerentes ao gerenciamento explícito de memória, alguns autores desenvolveram coletores de lixo que podem ser usados em programas C++. Boehm por exemplo desenvolveu um coletor de lixo para C++ bastante difundido que inclui suporte a finalizadores [19, 17]. Qualquer estrutura de dados alocada na memória heap pode ser registrada para finalização através da função `GC_register_finalizer`, que recebe como parâmetros um ponteiro para o objeto a ser finalizado, e um ponteiro para a função a ser invocada pelo coletor de lixo. Cada objeto pode ter apenas um único finalizador ativo. Ao ser invocado, o finalizador recebe como parâmetro um ponteiro para o objeto que está sendo finalizado. A ordem de invocação de finalizadores segue o sentido de eventuais referências entre objetos finalizáveis. Qualquer ciclo de referências contendo um ou mais objetos finalizáveis impede a finalização, e consequentemente, a coleta destes objetos (nesse caso a restrição de coleta não

tem qualquer relação com o algoritmo usado pelo coletor de lixo, e é decorrente simplesmente de uma decisão de implementação).

### 3.1.3 Modula-3

Modula-3 oferece um mecanismo de finalização baseado em referências fracas. O método `FromRef` recebe como parâmetros uma referência (normal) para um objeto qualquer alocado na memória heap, e opcionalmente, um callback a ser executado quando o coletor de lixo determinar que o objeto referenciado tornou-se inacessível. O callback sempre é executado antes do objeto associado ser efetivamente desalocado, recebendo como parâmetro uma referência para o próprio objeto. 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 dos mesmos. No entanto o conjunto-raiz em Modula-3 inclui, além de variáveis globais e valores nos registros de ativação ativos, todos os objetos acessíveis através de referências fracas com callbacks não nulos. Consequentemente, agregações de objetos são sempre finalizadas seguindo uma ordem correta, ou seja, o sentido das referências entre os objetos. Por outro lado, referências cíclicas impedem a finalização (e a coleta) de objetos finalizáveis.

O código abaixo, adaptado de [2], ilustra a implementação de finalizadores em Modula-3. Considere uma classe qualquer que envia dados através de um stream e utiliza um buffer. Ao tornar-se inacessível, um objeto desta classe deve esvaziar o buffer e fechar o stream correspondente. Essa dinâmica pode ser implementada da seguinte forma:

```
(* Define uma classe que representa um stream buferizado. *)
MODULE MyStream; IMPORT WeakRef, Wr, ...;

  PROCEDURE New(...): Stream =
    (* inicializa res como um Stream *)
    VAR res := NEW(Stream);
    BEGIN
      ...
      (* cria uma referência fraca e associa um callback para ela *)
      EVAL WeakRef.FromRef(res, Cleanup);
      RETURN res
  END New;
```

```
  ...

  (* callback finalizador *)
  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.
```

### 3.1.4 Java

Java segue o que poderíamos denominar de semântica padrão de finalização baseada em classes. A classe `java.lang.Object`, da qual todas as classes em Java são derivadas, implementa um método vazio denominado `finalize`, que não recebe nenhum parâmetro além da referência implícita ao próprio objeto (this). São considerados finalizáveis os objetos de qualquer classe que sobrescreve este método com um conjunto não vazio de instruções.

A invocação do finalizador acontece de forma implícita após o coletor determinar que o objeto correspondente é coletável, mas antes dele ser efetivamente coletado. Não existe qualquer restrição à ressurreição de objetos através de finalizadores, porém o finalizador associado a um objeto só é executado no máximo uma única vez, considerando-se inclusive invocações explícitas (finalizadores não podem ser reabilitados). Por razões que iremos discutir posteriormente, finalizadores nunca são invocados ao término da aplicação.

Java não oferece qualquer garantia quanto a ordem com que finalizadores são invocados, portanto os objetos acessados por um finalizador podem ter sido finalizados. Qualquer exceção lançada durante a execução de um finalizador que não é capturada de forma explícita pelo mesmo, interrompe a sua execução, mas não é propagada para a aplicação.

### 3.1.5 C#

A semântica básica de finalizadores no ambiente .Net é semelhante à de Java, mas a API associada a esta facilidade é muito mais extensa. Como em Java, todos os objetos de classes que definem um finalizador são considerados finalizáveis[1], e não são oferecidas quaisquer garantias em relação a quando ou em que ordem finalizadores serão invocados.

A princípio, o finalizador associado a um objeto também só é invocado uma única vez, sempre de forma implícita. Entretanto este comportamento pode ser alterado através dos métodos `GC.ReRegisterForFinalize` e `GC.SuppressFinalize`, que respectivamente reabilitam e desabilitam o finalizador associado a um objeto particular. Diferentemente de Java, todos os finalizadores que não forem explicitamente desabilitados são invocados antes do término da aplicação, e não é possível invocar explicitamente um finalizador.

Com o objetivo de permitir um maior controle sobre o momento em que o finalizador é executado, acomodando inclusive uma semântica semelhante à de destrutores C++, o ambiente .Net introduziu a interface `IDisposable`. Esta interface define um método único (`Dispose`) que deve ser invocado explicitamente pelo programa cliente para liberar de forma síncrona os recursos associados a um objeto. O exemplo seguinte implementa uma classe que representa arquivos, e ilustra a utilização de `IDisposable` em C#.

```
using System;
using System.IO;

/*
 * Esta classe representa um arquivo que pode ser finalizado
 * explicitamente através do método close. Se o programa
 * cliente não invocar o método close, o finalizador será
 * invocado automaticamente pelo coletor após o objeto
 * correspondente tornar-se inacessível.
 */
class MyFile: IDisposable{

  // indica se o método dispose já foi chamado
```

[1]Em C# finalizadores são definidos usando a mesma sintaxe de destrutores em C++. O finalizador é definido com o mesmo nome da classe base precedido pelo símbolo ∼, e não inclui um tipo de retorno.

```
private bool disposed = false;

// o recurso externo
private FileStream file;

// construtor
public MyFile(String fileName){
  file = new FileStream(fileName, FileMode.Open,
    FileAccess.Read, FileShare.Read);
}

// libera o recurso externo
public void Dispose(){
  if (!disposed){
    disposed = true;
    // libera o recurso externo
    file.Close();
    // evita que o objeto seja finalizado caso Dispose
    // já tenha sido chamado
    GC.SupressFinalize(this);
  }
}

// método que deve ser chamado para liberar o recurso externo
public void close(){
  Dispose();
 }

// finalizador
~MyFile(){
  Dispose();
}

// um método qualquer
public byte[] Read(){
  if (disposed)
    throw new ObjectDisposedException();
  return file.Read();
```

```
  }
}
```

Para facilitar a desaloção síncrona de recursos, C# oferece ainda a declaração `using`. O código

```
using (MyFile f = new MyFile("c:\tmp")) {
  byte[] b = f.Read();
}
```

é um açucar sintático para

```
MyFile f = new MyFile("c:\tmp");
try {
  byte[] b = f.Read();
} finally {
  f.Dispose();
}
```

Note que esse mecanismo cria uma dinâmica de execução semelhante à dinâmica de invocação de destrutores C++ para objetos criados na pilha, desde que referências criadas dentro da declaração *using* não sejam armazenadas em variáveis que permaneçam acessíveis após a execução do respectivo bloco de código. O manual do ambiente .Net recomenda o uso dessa estrutura sempre que for necessário garantir a finalização síncrona de objetos [9].

### 3.1.6 Smalltalk

Em Dolphin Smalltalk [8] podem ser finalizados apenas os objetos de classes que implementam o método `finalize`. Diferentemente de Java e C# entretanto, só são efetivamente elegíveis para finalização os objetos explicitamente registrados através do método `beFinalizable` (para desabilitar a finalização de um objeto basta invocar o método `beUnfinalizable`). A ordem de finalização dos objetos segue o sentido das referências entre os mesmos, e objetos referenciados por objetos finalizáveis não são coletados até que estes sejam finalizados.

### 3.1.7 Ruby

Em Ruby a definição de finalizadores é desacoplada da definição de classes. Para registrar um finalizador para um objeto específico deve-se utilizar o

método `ObjectSpace.define_finalizer` que recebe como parâmetros um objeto e um fechamento. É possível desabilitar a finalização através do método `ObjectSpace.undefine_finalizer`, e mais de um finalizador pode ser associado a um objeto.

O fechamento registrado como finalizador é armazenado em uma estrutura de dados estática (`ObjectSpace`), e portanto não pode incluir nenhuma referência para o objeto a ser finalizado (a existência de uma referência impede a finalização do objeto). Ao contrário do que acontece nas outras linguagens discutidas até aqui, em Ruby o finalizador é executado após o objeto associado ter sido coletado. Assim, qualquer informação necessária a finalização do objeto deve ser mantida pelo próprio fechamento.

### 3.1.8 Python

Finalizadores em Python seguem a semântica típica de linguagens orientadas a objetos: são acoplados a classes e definidos através da implementação de um método com uma assinatura padrão (`__del__`). Ainda que a especificação desta linguagem não ofereça qualquer garantia em relação ao momento exato em que acontece a invocação de finalizadores, a implementação atual de Python utiliza um coletor de lixo baseado em contagem de referências. Por conseguinte, finalizadores são invocados imediatamente após a contagem de referências atingir zero, ou imediatamente após a execução dos callbacks associados a referências fracas que referenciam o objeto finalizado, se estes callbacks existirem. A ordem de invocação dos finalizadores segue a ordem com que a contagem de referências dos objetos chega a zero. Assim como em Java, não são oferecidas quaisquer garantias em relação à invocação de finalizadores antes do término de um programa.

Python oferece ainda um segundo mecanismo de finalização, baseado em referências fracas. Um callback pode ser associado a qualquer referência fraca, sendo invocado assim que o objeto referenciado tornar-se inalcançável (mas antes de ser coletado, ou mesmo finalizado). A principal diferença entre finalizadores ordinários e callbacks é que este último pode ser associado dinamicamente a uma instância de uma classe, enquanto o primeiro é estaticamente associado a todas as instâncias de uma classe. Além disso pode-se associar múltiplos callbacks a um objeto qualquer.

### 3.1.9 Perl

Finalizadores em Perl são definidos apenas por pacotes (a abstração da linguagem usada para representar classes), e são invocados imediatamente após o objeto tornar-se inacessível (o coletor de lixo de Perl também usa contagem de referências). Os finalizadores são invocados antes do término do programa, mas neste caso não existem garantias quanto a ordem da invocação.

### 3.1.10 Lua

Lua 5.0 suporta um mecanismo de finalização que pode ser usado exclusivamente com um tipo específico (`userdata`), que representa dados arbitrários em C instanciados pelo programa hospedeiro ou por uma biblioteca. O finalizador deve ser registrado para cada objeto a ser finalizado (através da definição do metamétodo `_gc`). Após o coletor de lixo determinar que um objeto não pode ser mais acessado pelo programa Lua, ele insere o finalizador correspondente em uma fila. Ao final do ciclo de coleta de lixo os finalizadores são invocados na ordem inversa da criação dos objetos (ordem de elaboração), recebendo o próprio objeto como parâmetro. Assim como Perl e C#, Lua garante que todos os finalizadores serão invocados antes do término da aplicação.

## 3.2 Referências Fracas

Nesta seção descrevemos as principais interfaces associadas a referências fracas e às suas extensões.

### 3.2.1 Java

Java oferece três tipos distintos de referências fracas: *suave* (soft), *fraca* (weak) e *fantasma* (phantom) [52]. Cada uma destas referências é implementada por uma subclasse da classe abstrata `Reference`, e representam conjuntamente gradações sucessivas de referências:

- Um objeto é *fortemente conexo* (strongly reachable) se ele pode ser alcançado através de uma cadeia de referências ordinárias a partir da pilha de execução ou de variáveis globais (o conjunto-raiz).

- Um objeto é *suavemente conexo* (softly reachable) se ele não é fortemente conexo, e pode ser alcançado a partir do conjunto-raiz através de uma cadeia de referências composta por referências ordinárias e por pelo menos uma referência suave.

- Um objeto é *fracamente conexo* (weakly reachable) se ele não é fortemente nem suavemente conexo, e pode ser alcançado a partir do conjunto-raiz através de uma cadeia de referências composta por pelo menos uma referência fraca, e por nenhuma referência fantasma.

- Um objeto é um *fantasma* (phantom reachable) se ele não é fortemente, suavemente e nem fracamente conexo, pode ser alcançado a partir do conjunto-raiz através de uma cadeia de referências composta por pelo menos uma referência fantasma, e caso tenha um finalizador, o mesmo já tenha sido executado.

Referências suaves e fracas são idênticas do ponto de vista semântico. A única diferença prática entre as mesmas, e que depende da implementação da máquina virtual, é o momento em que são coletadas. Uma referência fraca *será* limpa tão logo o coletor de lixo determine que o objeto referenciado é fracamente conexo, enquanto uma referência suave *poderá* ser limpa tão logo o coletor de lixo determine que o objeto referenciado é suavemente conexo. Na prática, objetos suavemente conexos só são coletados quando a utilização da memória alcança níveis relativamente altos. Em ambos os casos, enquanto as referências não forem limpas, os objetos referenciados podem ser acessados pelo programa cliente.

Referências fantasmas constituem na verdade um mecanismo alternativo de finalização, que permite tratar recursos computacionais externos ao ambiente Java. Por si só esse tipo de referência é inútil, já que não permite acesso ao objeto referenciado. Seu uso deve ser sempre acoplado a classe auxiliar `ReferenceQueue` (que pode ser associada a qualquer instância de subclasses de `Reference`). Para poder determinar se um objeto virou um fantasma, o programa cliente deve registrar a referência fantasma associada ao objeto numa `ReferenceQueue`, e periodicamente efetuar uma verificação na mesma. O coletor de lixo irá adicionar à fila todas as referências que foram limpas.

Java oferece suporte também a arrays associativos fracos (`WeakHashMap`). A implementação desta estrutura de dados é composta por uma tabela hash ordinária (`HashMap`) e uma `ReferenceQueue`. Sempre que um par chave/valor é inserido em um `WeakHashMap`, a chave é automaticamente encapsulada em uma

referência fraca e é inserida, juntamente com o valor original, no `HashMap`. As referência fracas que representam chaves são registradas ainda na `ReferenceQueue`. Sempre que um método do `WeakHashMap` é chamado, a `ReferenceQueue` é checada, e as entradas cujas chaves foram limpas são implicitamente removidas da tabela.

### 3.2.2 Eiffel

Em SmartEiffel (também conhecido como GNU Eiffel) referências fracas são implementadas através de classes paramétricas (templates), o que permite que o compilador efetue uma verificação de tipos mais completa, e consiga otimizar melhor expressões contendo objetos dessas classes. O suporte a referências fracas é ainda mais amplo do que em Java, e permite inclusive que o programador defina explicitamente a dinâmica de coleta de objetos fracamente referenciados.

Referências fracas são implementadas através das seguintes classes:

- `WEAK_REFERENCE{G}` define referências fracas ordinárias (`G` representa o tipo do objeto a ser referenciado). Para referenciar um objeto através de uma instância qualquer desta classe usa-se o método `set_item`. Um objeto fracamente referenciado pode ser diretamente acessado através do atributo `item`.

- `SOFT_REFERENCE{G}` define referências suaves, cuja semântica é similar às referências suaves de Java (o objeto fracamente referenciado só é coletado quando o nível de utilização da memória for significativo).

- `TUNNABLE_STRENGTH_REFERENCE{G}` representa referências cujo grau de fraqueza é definido explicitamente pelo programador (através dos métodos `set_item_with_strength` e `set_strength`). O coletor irá coletar os objetos referenciados por referências menos fracas somente quando a memória disponível para a aplicação tornar-se reduzida. Essa coleta é realizada de forma ordenada, seguindo o grau de fraqueza das referências (em cada ciclo as referências mais fracas são coletadas primeiro, e somente se necessário, referências menos fracas são também coletadas).

- `PROGRAMMABLE_REFERENCE{G}` é uma classe abstrata que permite ao programador especificar em detalhes quando objetos fracamente referenciados devem ser coletados. Antes de reclamar qualquer objeto referenciado por

uma instância de uma subclasse de `PROGRAMMABLE_REFERENCE{G}`, o coletor invoca o método booleano `item_reclamation_allowed`. O objeto só é coletado, e a referência fraca só é limpa, se o valor retornado for verdadeiro.

### 3.2.3 C#

O ambiente .Net suporta duas formas de referências fracas, *referências curtas* e *referências longas*, ambas representadas pela classe `WeakReference`. Referências curtas são limpas logo após o coletor determinar que o objeto referenciado é fracamente conexo. Referências longas só são limpas após o objeto referenciado ter sido finalizado. Para classes que não possuem finalizadores, referências curtas e longas comportam-se de forma idêntica.

Como o ambiente .Net não impõe qualquer restrição de acesso a objetos referenciados por referências fracas, referências longas permitem a ressurreição de objetos finalizados (em Java isso não é possível porque referências fantasmas não oferecem acesso ao objeto referenciado).

### 3.2.4 Python

Em Python referências fracas são implementadas pelo módulo `weakref`, o qual possui uma API relativamente extensa. A forma mais simples de se criar uma referância fraca é através da função `ref`, que recebe o objeto a ser referenciado como parâmetro, e retorna um novo objeto do tipo `WeakReference`. Como parâmetro opcional, essa função aceita um callback que é invocado após a referência ser limpa. O callback por sua vez recebe como único parâmetro a referência que foi limpa. Se mais de um callback estiver associado a um objeto, os callbacks serão executados na ordem inversa em que foram registrados.

Uma forma alternativa de criar uma referência fraca em Python é através da função `proxy`, que também recebe como parâmetro o objeto a ser referenciado, mas retorna um proxy (objetos do tipo `ProxyType` ou `CallableProxyType`) que pode ser usado como o próprio objeto sem a necessidade de dereferenciação (a dereferenciação é implícita). Como objetos proxy podem ser limpos a qualquer momento, a especificação da linguagem não permite que os mesmos sejam usado como chaves de arrays associativos.

O módulo `weakref` inclui ainda dois tipos diferentes de tabelas fracas. A classe `WeakKeyDictionary` representa uma tabela hash com chaves fracas e valores

fortes, enquanto a classe `WeakValueDictionary` representa uma tabela com chaves fortes e valores fracos. Em ambos os casos os elementos são automaticamente removidos da tabela após a referência fraca (a chave ou o valor, dependendo do tipo de tabela) ser limpa.

### 3.2.5 Smalltalk

GNU Smalltalk [21] oferece suporte a diversos tipos de coleções fracas, como conjuntos, tabelas hash e até mesmo arrays. Conjuntos fracos suportam todas as operações típicas de conjuntos, e membros do conjunto são automaticamente removidos após o coletor determinar que a sua conectividade é fraca.

Como característica mais marcante da semântica desta linguagem, qualquer objeto pode ser transformado em um *objeto fraco* através da mensagem `makeWeak`. Um objeto fraco retém o seu tipo original, mas todos os seus atributos tornam-se referências fracas. Assim como acontece com qualquer referência fraca ordinária, o coletor de lixo limpa os atributos antes de coletar os objetos fracamente referenciados pelos mesmos.

### 3.2.6 Perl

Referências fracas foram introduzidas na distribuição padrão de Perl a partir da versão 5.8, com o objetivo principal de contornar o problema de coleta de referências cíclicas (conforme já mencionamos, esta linguagem utiliza um coletor de lixo baseado em contagem de referências). Perl permite criar referências fortes para quaisquer variáveis, subrotinas, e mesmo valores, através do operador \. Qualquer referência forte pode ser transformada em fraca através da função `weaken`. Assim que a contagem de referências associada a um objeto fracamente referenciado atinge zero, todas as referências fracas correspondentes passam a apontar para o valor `undef` (equivalente a `null`). A função `isweak` permite testar se uma referência qualquer é fraca, mas não existe nenhuma função para transformar uma referência fraca em uma referência normal.

### 3.2.7 Lua

Lua 5.0 implementa referências fracas usando a principal estrutura de dados suportada pela linguagem: arrays associativos (conhecidos em Lua como *tabelas*

*Lua*). Usando metadata, o programador pode criar uma tabela em que apenas a chave é fraca, apenas o valor é fraco, ou ambos são fracos. Se qualquer campo fraco tornar-se coletável, o elemento correspondente é implicitamente removido da tabela.

### 3.2.8 Haskell

Glasgow Haskell (GHC) oferece uma implementação interessante de referências fracas [42]. Nesta linguagem uma referência fraca é representada por um par chave/valor, onde o valor é considerado acessível se a chave for acessível, mas a conectividade da chave não tem qualquer relação com a conectividade do valor, ou com a conectividade da própria referência fraca (o par chave/valor)[2]. Essa especificação é parecida com a semântica de ephemerons [35], a ser discutida no Capítulo 4, e evita que referências cíclicas entre chaves e valores impeçam a coleta dos elementos que compõe a referência fraca. Isso é importante principalmente na implementação de tabelas fracas.

GHC permite também que sejam associadas ações (basicamente callbacks, que em GHC são denominados finalizadores) a referências fracas, as quais são executadas após as chaves serem limpas. Se múltiplos finalizadores forem associadas a mesma chave, estas serão executadas em uma ordem arbitrária, ou mesmo de forma concorrente.

Finalizadores podem incluir explicitamente referências para as chaves a serem finalizadas, mas tais referências não impedem a sua efetiva finalização, já que são tratadas exatamente como um valor de um ephemeron (o valor e *o finalizador* são considerados acessíveis apenas se a chave for acessível, mas a conectividade da chave não tem qualquer relação com a conectividade da referência fraca, do valor ou do finalizador associado). GHC garante ainda que finalizadores registrados serão executados uma única vez, quer seja quando o ephemeron correspondente for limpo, ou através de uma invocação explícita, ou ainda ao final da execução da aplicação.

[2]Em linguagens puramente funcionais, efeitos colaterais e estados globais podem ser representados através de *monads*. O leitor interessado neste conceito pode consultar [67].

### 3.2.9 Modula-3

Em Modula-3 referências fracas são criadas através do método `FromRef`, o qual recebe como parâmetros uma referência para um objeto qualquer alocado na memória heap, e opcionalmente um callback a ser executado quando o coletor de lixo determinar que o objeto fracamente referenciado tornou-se inacessível para o programa cliente. A referência fraca é o valor retornado por `FromRef`. A função `ToRef` testa uma referência fraca e retorna uma referência normal para o objeto referenciado, caso o objeto seja considerado acessível, ou `nil`, caso contrário. O callback recebe como parâmetro uma referência para o objeto que tornou-se inacessível (ao invés da referência fraca), podendo portanto ressuscitá-lo.

## 3.3 Considerações Finais

Como pode ser facilmente percebido, o suporte tanto a finalizadores quanto a referências fracas varia bastante entre as diferentes linguagens de programação consideradas aqui. Tal variação pode ser explicada pelas diferenças naturais entre essas linguagens, e sobretudo, na nossa opinião, por uma relativa falta de consenso quanto às características que são efetivamente importantes no suporte a esses mecanismos. No próximo capítulo iremos efetuar uma análise mais profunda destas questões semânticas.