3
Compilando Bytecodes Lua para CIL

Este capítulo descreve a segunda abordagem de integração entre a Linguagem Lua e o Common Language Runtime, através da criação de um compilador escrito em C#, chamado Lua2IL, para traduzir os bytecodes da máquina virtual Lua 5.0 em instruções da Common Intermediate Language do CLR.

O compilador emite código CIL independente do interpretador Lua. Ele permite tanto a execução de scripts Lua pelo CLR quanto a interoperabilidade entre os scripts e os objetos do CLR. A interoperabilidade abrange tanto a execução de funções Lua por programas do CLR quanto a instanciação e uso de objetos CLR por scripts Lua. O nível de interoperabilidade é o mesmo da biblioteca LuaInterface, apresentada no Capítulo 2.

As seções seguintes descrevem a máquina virtual de Lua 5.0, o mapeamento dos tipos Lua para tipos do CTS, como Lua2IL traduz os bytecodes para código CIL, e a implementação de outros recursos de Lua não diretamente explícitos nos bytecodes: co-rotinas e weak tables. A última seção trata da interface entre o código gerado e o resto do CLR.

3.1
A Máquina Virtual de Lua 5.0

O interpretador Lua 5.0 implementa uma máquina virtual baseada em registradores, com 35 instruções de tamanho fixo. As instruções recebem até três argumentos; o primeiro sempre é um registrador, e os outros podem ser registradores, constantes ou mesmo valores imediatos, dependendo da instrução. O Apêndice A lista todas as 35 instruções da máquina virtual de Lua 5.0.

Cada função possui seu próprio conjunto de registradores, começando pelo registrador 0, com um número variável deles (limitado a 256 registradores). Os registradores são mapeados para posições na pilha de execução do interpretador, a partir da posição base da pilha para a função. O topo

da pilha geralmente é usado por instruções que operam sobre um número variável de registradores.

As funções recebem seus argumentos a partir do registrador 0, na ordem em que eles ocorrem na chamada à função. Suas variáveis locais e valores de retorno também são armazenados nos registradores. As funções também possuem um vetor com as constantes da função, uma tabela para suas variáveis globais (por padrão a mesma da função que definiu a função corrente) e um vetor de *upvalues*. Os upvalues são referências para variáveis locais de escopos externos ao escopo corrente. Um upvalue começa *em aberto*: como uma referência para a posição da variável correspondente, na pilha de execução. Quando o escopo dessa variável é abandonado (e ela é consequentemente descartada da pilha) o upvalue é fechado: a referência é substituída pelo próprio valor da variável. Duas funções que usam a mesma variável local externa compartilham um mesmo upvalue, portanto elas continuam usando a mesma variável mesmo quando o upvalue é fechado.

A máquina virtual também define uma série de ganchos para depuração e instrumentação por outros programas. O funcionamento destes ganchos foge ao escopo deste trabalho; a sua implementação por Lua2IL seria redundante, já que o CLR oferece seus próprios mecanismos para depuração e instrumentação de código.

3.2 Mapeamento dos Tipos de Lua para o CTS

Lua2IL representa todos os valores de Lua por instâncias da estrutura `LuaValue`. Esta estrutura possui dois campos, um campo chamado `O`, de tipo `LuaReference` e um campo chamado `N`, de tipo `double`. Se o o primeiro campo for `null` o valor é um número, armazenado no segundo campo; caso contrário o valor é o objeto armazenado no primeiro campo. `LuaReference` define um campo *tag* e métodos virtuais para as operações feitas sobre os valores. Existem métodos virtuais para as operações de indexação, chamada ao valor e comparação com outros valores.

Como visto no parágrafo anterior, o tipo número é mapeado para o tipo `double`, armazenado dentro de um dos campos de `LuaValue`. Os outros tipos de Lua são mapeados para subclasses de `LuaReference`. As strings são mapeadas para instâncias de `LuaString`, as tabelas para instâncias de `LuaTable`, o valor booleano `true` para a instância única de `TrueClass`, o valor booleano `false` para a instância única de `FalseClass` e o valor `nil` para a instância única de `NilClass`. O tipo userdata é mapeado

para instâncias da classe `LuaWrapper`, e estas instâncias são proxies para instâncias de tipos do CTS; mais detalhes desse mapeamento estão na Seção 3.5.

As instâncias de `LuaString` guardam o valor da cadeia de caracteres em uma instância de `System.String`. As instâncias de `LuaTable` guardam os valores da tabela em uma instância da classe `System.Collections.Hashtable`, uma classe da biblioteca padrão do CLR que implementa tabelas hash. A classe `LuaTable` e as subclasses de `LuaReference` definem as operações necessárias (códigos hash e comparação) para que as tabelas hash funcionem corretamente quando indexadas por qualquer valor Lua.

Lua2IL mapeia cada função definida em um script para uma subclasse de `LuaClosure`, redefinindo seu método virtual `Call` para executar o código da função (a tradução em IL dos seus bytecodes). O método `Call` é o método virtual da classe `LuaReference` responsável pela operação de chamada ao valor; por padrão ele gera uma exceção, indicando que o valor não provê esta operação. O corpo principal de um script também é uma função; ele é mapeado para uma subclasse de `LuaClosure` chamada `MainFunction`. Para executar um script compilado por Lua2IL, um programa CLR instancia esta classe e chama o seu método `Call`.

A razão de não mapear diretamente os tipos de Lua número, string e booleano para seus correspondentes no CTS é o desempenho do código gerado. As variáveis e parâmetros formais de funções Lua não têm tipos declarados; o mapeamento direto dos tipos exigiria que o código tratasse todos os valores como do tipo `object`. A consequência seriam testes de tipo e casts a cada operação.

O mapeamento para novos tipos permite usar outro tipo como denominador comum (no caso, `LuaValue` e `LuaReference`), eliminando a necessidade de testes de tipo e casts. Para verificar se uma instância de `LuaValue` é um número, por exemplo, basta ver se o seu campo `O` é `null`. Sendo um número, o seu valor é obtido lendo o seu campo `N`. Como outro exemplo, para indexar uma instância de `LuaValue` basta ver se o seu campo `O` não é `null`. Não sendo, basta chamar o método de indexação da classe `LuaReference` no campo. Se o objeto não prover esta operação, Lua2IL gera uma exceção.

3.3 Compilação das Instruções

Para compilar um arquivo de bytecodes, Lua2IL lê o arquivo para a memória, em uma estrutura em árvore contendo todas as informações que estavam no arquivo. Cada nó da árvore é uma função, começando pelo corpo. Lua2IL caminha por essa árvore, em pré-ordem, compilando cada nó em uma subclasse de `LuaClosure`. O resultado final é uma biblioteca (uma *assembly* do CLR) com o mesmo nome do arquivo original, contendo todas as classes geradas.

A execução do código gerado para cada função usa uma pilha de execução, similar à do interpretador Lua 5.0. A pilha é usada para guardar variáveis locais, passar argumentos para funções e retornar valores. Para que outros programas do CLR tenham uma interface mais natural para chamar as funções Lua, a classe `LuaClosure` define um método utilitário que recebe os argumentos em um vetor, os empilha, e chama a função; depois desempilha os valores de retorno e os retorna em outro vetor.

Lua2IL usa um vetor de valores do tipo `LuaValue` para implementar a pilha. Quando a pilha precisa crescer além da última posição um novo vetor é alocado, com o dobro do tamanho do anterior; os valores então são copiados de um vetor para outro, e o novo vetor toma o lugar do antigo. As funções geradas pelo compilador também mantêm as estruturas presentes no interpretador Lua: a tabela de variáveis globais, o vetor de constantes e o vetor de upvalues. Também é mantida uma lista ligada de upvalues em aberto.

A compilação da maioria das instruções da máquina virtual é direta, pois as estruturas principais do interpretador Lua estão duplicadas no código gerado por Lua2IL. O código CIL destas instruções é uma tradução direta do código C do interpretador, levando-se em conta as diferenças nas estruturas de dados envolvidas.

Para compilar as instruções que envolvem saltos, Lua2IL faz uma primeira passada pelos bytecodes, e calcula os destinos de cada salto, guardando-os em uma tabela. Na compilação da instrução OP_JMP, Lua2IL consulta a tabela e emite uma instrução de salto da CIL para o destino correto.

As instruções OP_CALL e OP_TAILCALL são as que mais divergem de suas implementações no interpretador Lua, já que Lua2IL não precisa criar registros de execução para cada chamada de função, deixando esta tarefa para o CLR. O método `Call` de cada função (que contém o código

gerado por Lua2IL para a função) recebe, como argumentos, um objeto contendo a pilha e a lista de upvalues em aberto, o número de valores de retorno esperados e a posição do seu registrador 0.

A instrução OP_CALL apenas marca a posição do último argumento na pilha e chama o método `Call` do valor armazenado no registrador A. Um preâmbulo dentro do método `Call` ajusta os argumentos na pilha para a quantidade de argumentos que a função espera e limpa o espaço na pilha que vai ser usado pela função (possivelmente aumentando a pilha). O código que segue este preâmbulo já é o código da própria função. OP_TAILCALL faz a cópia dos argumentos da função chamada para os registradores da função atual, a partir do registrador 0, e depois chama a função. A CIL possui uma instrução que chama um método reaproveitando o registro de execução do método chamador, mantendo o comportamento esperado de OP_TAILCALL.

O primeiro protótipo do compilador emitia, para cada instrução, uma chamada a um método de `LuaClosure` que recebia os argumentos da instrução e a executava. Os métodos eram implementados em C#, para facilitar a depuração. Depois que todas as instruções foram implementadas e o comportamento do compilador verificado, o compilador foi modificado para melhorar o desempenho; para cada instrução ele passou a emitir o equivalente CIL do código C# que implementa a instrução.

A mudança permitiu que todos os testes envolvendo o valor de um dos argumentos das instruções, presentes em metade das instruções e que antes eram feitos durante a execução, sejam feitos durante a compilação. O vetor de constantes foi abandonado: o código gerado para a instrução OP_LOADK é especializado para a constante que se está carregando, determinado em tempo de compilação. O vetor de upvalues também foi abandonado: cada upvalue passa a ser um campo no objeto da função, já que as posições no vetor também são todas determinadas em tempo de compilação.

Algumas instruções ainda são implementadas parcial ou totalmente pelo runtime Lua2IL, em C#. O código CIL para as instruções aritméticas trata apenas do caso em que ambos os operandos são números. Nos outros casos o código chama um método do runtime Lua2IL que testa se os valores possuem a operação apropriada, delegando a execução para ela. As instruções de comparação são compiladas da mesma forma. Instruções com operação variável, a depender do tipo do valor sobre o qual elas estão operando, também chamam métodos do runtime Lua2IL durante sua execução.

3.4 Implementando Outros Recursos de Lua 5.0

As instruções da máquina virtual não cobrem todos os recursos da linguagem Lua 5.0. Co-rotinas são implementadas por funções da biblioteca padrão da linguagem Lua. Weak tables são parte da implementação do coletor de lixo. Apenas traduzir as instruções da máquina virtual para CIL não provê estes recursos. Esta seção descreve como eles são implementados por Lua2IL.

3.4.1 Co-rotinas

Co-rotinas são linhas independentes de execução dentro de um script Lua, mas, ao contrário das *threads* do CLR, as co-rotinas não possuem um escalonador: a execução de uma co-rotina só é suspensa quando ela chama uma função para isso. Em um dado momento apenas uma co-rotina pode estar executando, mesmo que a máquina seja capaz de execução paralela.

Um script Lua cria uma co-rotina com a função `coroutine.create`. A função recebe a função principal da co-rotina como argumento, e retorna um objeto que representa a nova co-rotina. Para iniciar a execução de uma co-rotina chama-se a função `coroutine.resume`, passando a co-rotina e os argumentos para ela.

Dentro de uma co-rotina, a função `coroutine.yield` suspende a sua execução. A chamada a essa função pode ser em qualquer ponto da execução da co-rotina, não necessariamente dentro da sua função principal. A chamada a `coroutine.yield` faz a chamada a `coroutine.resume` que iniciou a co-rotina retornar. A função `coroutine.resume` retorna `true` seguido dos argumentos para `coroutine.yield`.

Chamar `coroutine.resume` uma segunda vez, passando a mesma co-rotina, retoma a execução da co-rotina no ponto onde ela parou, retornando da chamada a `coroutine.yield`. A chamada a `coroutine.yield` retorna os argumentos passados para `coroutine.resume`. Quando a função principal da co-rotina termina a chamada a última chamada a `coroutine.resume` retorna `true` seguido dos valores de retorno da função principal. Durante a execução da co-rotina, caso aconteça qualquer erro que não seja capturado a chamada a `coroutine.resume` retorna `false` e o erro.

Lua2IL implementa co-rotinas usando *threads* do CLR e semáforos. Cada co-rotina tem associada a ela uma pilha de execução própria, uma thread e dois semáforos binários. Os semáforos são *resume* e *yield*, e são

criados com estado 0 (fechados). Quando uma co-rotina é criada a thread que a criou inicia a thread da co-rotina; a thread da co-rotina tenta decrementar o semáforo *resume* e é suspensa.

Quando a thread principal chama `coroutine.resume` ela copia os argumentos para a pilha da co-rotina, incrementa o semáforo *resume* e tenta decrementar o semáforo *yield*. A co-rotina é liberada para continuar a execução e a thread principal é suspensa.

Quando a co-rotina chama `coroutine.yield` ela incrementa o semáforo *yield* e tenta decrementar o semáforo *resume*. A co-rotina é suspensa e a thread principal continua a execução, transportando os argumentos de `coroutine.yield` para a sua pilha. Quando a co-rotina termina ela também incrementa o semáforo *yield*, e marca um flag que a impede de ser executada novamente.

Se uma exceção ocorre durante a execução da co-rotina, o runtime Lua2IL captura a exceção, a empilha como valor de retorno da co-rotina, e encerra a co-rotina. Antes de encerrá-la todos os upvalues em aberto na pilha da co-rotina são fechados. A função `coroutine.resume` retorna o erro quando a execução volta para a thread principal.

A implementação de Lua2IL para co-rotinas reproduz exatamente o comportamento das co-rotinas do interpretador Lua, mas é ineficiente; os semáforos, necessários para forçar que as threads se comportem de maneira síncrona, são custosos. Mas esta é, no momento, a única maneira no CLR de se implementar co-rotinas em código gerenciado. Uma implementação que envolve código nativo existe, mas depende de uma biblioteca específica da plataforma Windows, e sua interação com o sistema de exceções e o coletor de lixo do CLR é problemática [21].

3.4.2 Weak Tables

Uma weak table é uma tabela cujos elementos são referências fracas. Uma referência fraca não conta como referência para o coletor de lixo: se existem apenas referências fracas para um objeto ele é coletado. Uma weak table pode ter chaves fracas, valores fracos, ou ambos. Se uma chave ou um valor é coletado então o par é removido da tabela.

No interpretador Lua 5.0, as weak tables são implementadas pelo coletor de lixo: antes de começar a marcar as referências em uma tabela, o coletor verifica se ela tem referências fracas (se ela foi marcada pelo script como uma weak table). As tabelas com referências fracas são postas em

listas; ao fim de fase de marcação o coletor remove das tabelas todos os pares com referências fracas não marcadas.

Lua2IL implementa weak tables usando as referências fracas do CLR. Ao armazenar um valor em uma tabela com valores fracos, o runtime Lua2IL empacota o valor dentro de uma instância de `System.WeakReference`. Se a tabela tem chaves fracas, o runtime Lua2IL empacota a chave dentro de uma instância de `WeakReference` antes de indexar a tabela.

Para garantir que a tabela funcione corretamente, o código hash de uma instância de `WeakReference` deve ser o mesmo do objeto para o qual ele aponta. Além disso, a comparação entre instâncias de `WeakReference` deve ser feita comparando-se os seus objetos referenciados. A classe `WeakReference` não implementa nenhum dos dois comportamentos, entretanto. O runtime Lua2IL então define um gerador de códigos hash específico para as weak tables com chaves fracas; estas tabelas também têm um comparador específico que compara os objetos apontados pelas instâncias de `WeakReference` ao invés das instâncias em si.

A implementação de Lua2IL para weak tables é ineficiente, se comparada com a do interpretador Lua. Ela introduz custos na indexação das weak tables; o custo de criar as referências fracas e de verificar se elas ainda são válidas nas operações da tabela (obter código hash e comparar chaves). Além da ineficiência, um problema da implementação atual é a permanência, na tabela, de referências fracas apontando para objetos já coletados. A tabela não é notificada quando uma chave, ou um valor, é coletada, logo ela não remove o par. O único evento associado à coleta de lixo, no CLR, é na finalização de um objeto, quando o seu método `Finalize` é chamado. As referências fracas não são notificadas quando se tornam inválidas, nem há como registrar um método para ser chamado ao fim de um ciclo de coleta de lixo.

Na máquina virtual da linguagem Java (a JVM), em comparação, pode-se associar uma fila a uma referência fraca: quando o objeto apontado pela referência fraca é coletado o coletor de lixo pôe a referência na fila. Com este recurso é possível implementar weak tables como as de Lua: cada tabela tem sua fila de referências coletadas, e uma thread responsável por percorrer esta fila remove da tabela as referências inválidas.

3.5 Interface com o CLR

Lua2IL oferece a mesma interface com objetos do CLR que a biblioteca LuaInterface. Lua2IL é, portanto, um consumidor completo, e extensor parcial, da Common Language Specification. A operação desta interface já foi discutida no Capítulo 2, mais especificamente na Seção 2.1, e não vai ser revista aqui.

O runtime Lua2IL é o responsável pela integração entre o código Lua e o restante do CLR. A classe `LuaWrapper`, descendente de `LuaReference`, representa os objetos do CLR manipulados pelo código Lua. `LuaWrapper` tem duas subclasses; uma representa tipos do CTS, e é responsável pela instanciação de objetos e acesso a membros estáticos. A outra subclasse de `LuaWrapper` representa as instâncias dos tipos, consequentemente responsável pelo acesso a membros de instância. A classe `LuaWrapper` e suas subclasses redefinem os métodos de `LuaReference` responsáveis pelas operações de indexação, atribuição e chamada.

Por exemplo, em uma chamada como `obj:foo(arg1,arg2)`, ou seja, `obj["foo"](obj,arg1,arg2)`, a operação `obj["foo"]` emite uma instrução OP_GETTABLE. O código que Lua2IL gera para esta instrução chama um método de indexação em `obj`. Se `obj` for uma instância de `LuaWrapper`, o seu método de indexação procura, usando a API de reflexão do CLR, por um método `foo` no objeto CLR representado por `obj`. Se for encontrado, o método de indexação retorna um *proxy* para este método. Se nenhum método for encontrado ele retorna `nil`.

Continuando o exemplo, anterior, a chamada ao valor retornado por `obj["foo"]`, `obj["foo"](obj,arg1,arg2)`, emite uma instrução OP_CALL (ou OP_TAILCALL). O código gerado por Lua2IL para esta instrução chama o método `Call` do valor retornado, ou seja, do proxy para o método. O proxy consulta, na pilha de execução, os argumentos para a chamada, faz a conversão dos argumentos para os tipos que o método exige, e chama o método. Se o método for sobrecarregado o proxy tenta chamar cada um dos métodos, na ordem em que são definidos; se nenhuma chamada for bem-sucedida é porque não existe uma versão compatível, e uma exceção é gerada.

O custo de uma busca reflexiva por um método é alto. Logo, para pagar este custo uma única vez, as instâncias de `LuaWrapper` mantêm os proxies em um cache, compartilhado por todas as instâncias de um mesmo tipo CTS. No caso de métodos sobrecarregados, o próprio proxy mantém

um cache com o último método chamado com sucesso; na próxima chamada o proxy tenta chamar o método em cache primeiro.

Voltando ao exemplo de `obj["foo"]`, se `foo` for um campo, o método de indexação de `obj` encontra `foo`, usando reflexão, retornando o valor de `foo` no objeto CLR representado por `obj`. O campo é armazenado em um cache, novamente para não precisar de outra busca reflexiva. Propriedades e eventos são tratados de forma análoga.

Na atribuição de um valor a um campo, como em `obj.foo=bar`, a atribuição emite uma instrução OP_SETTABLE. O código que Lua2IL gera para esta instrução chama o método de atribuição de `obj`. Este método acha o campo `foo`, usando reflexão, converte `val` para o tipo do campo, e faz a atribuição. O campo `foo` é armazenado no mesmo cache mencionado no parágrafo anterior. A atribuição a propriedades é feita de forma análoga.

A interface do código gerado por Lua2IL com o CLR também converte funções Lua para delegates automaticamente, como em LuaInterface. A implementação é similar à descrita na Seção 2.2.6. A criação de novas classes, descrita na Seção 2.1.7, também é implementada como em LuaInterface.