PUC-Rio - Certificação Digital Nº 0210489/CA

Fabio Mascarenhas de Queiroz

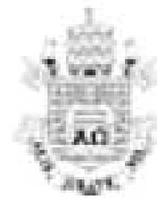
Integração entre a Linguagem Lua e o Common Language Runtime

DISSERTAÇÃO DE MESTRADO

DEPARTAMENTO DE INFORMÁTICA
Programa de Pós–graduação em Informática

Rio de Janeiro
Março de 2004

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO



Fabio Mascarenhas de Queiroz

Integração entre a Linguagem Lua e o Common Language Runtime

Dissertação de Mestrado

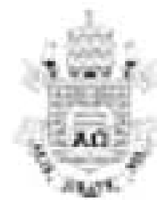
Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC

Orientador: Prof. Roberto Ierusalimschy

Rio de Janeiro
Março de 2004

PUC-Rio - Certificação Digital Nº 0210489/CA

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO



Fabio Mascarenhas de Queiroz

Integração entre a Linguagem Lua e o Common Language Runtime

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC.Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Roberto Ierusalimschy
Orientador
Departamento de Informática — PUC

Prof. Noemi Rodriguez
PUC–Rio

Prof. Renato Cerqueira
PUC–Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC

Rio de Janeiro, 19 de Março de 2004

PUC-Rio - Certificação Digital Nº 0210489/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Fabio Mascarenhas de Queiroz

Graduou-se Bacharel em Ciência da Computação pela Universidade Federal da Bahia (Salvador, Bahia).

PUC-Rio - Certificação Digital Nº 0210489/CA

Ficha Catalográfica

Queiroz, Fabio Mascarenhas de

Integração entre a Linguagem Lua e o Common Language Runtime/ Fabio Mascarenhas de Queiroz; orientador: Roberto Ierusalimschy. — Rio de Janeiro : PUC, Departamento de Informática, 2004.

v., 68 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Teses. 2. Common Language Runtime. 3. Compiladores. 4. Linguagem Lua. 5. Linguagens de programação. 6. Microsoft .NET. 7. Máquinas virtuais. I. Ierusalimschy, Roberto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

PUC-Rio - Certificação Digital Nº 0210489/CA

Ao meu orientador, Professor Roberto Ierusalimschy, pelo apoio ao trabalho, a disponibilidade para responder minhas dúvidas e a paciência para revisar os meus textos e apontar os meus erros. Agradeço também a ele e ao Professor Renato Cerqueira pela inspiração para este trabalho.

À CAPES e à PUC–Rio, pelo auxílio concedido, sem o qual a realização deste trabalho seria bem mais difícil.

À minha família, pelo apoio incondicional à minha decisão de me mudar para outro estado, para continuar a minha formação.

Resumo

Queiroz, Fabio Mascarenhas de; Ierusalimschy, Roberto. **Integração entre a Linguagem Lua e o Common Language Runtime**. Rio de Janeiro, 2004. 68p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

O Common Language Runtime (CLR) é uma plataforma criada com o objetivo de facilitar a interoperabilidade entre diferentes linguagens de programação, através de uma linguagem intermediária (a Common Intermediate Language, ou CIL) e um sistema de tipos comum (o Common Type System, ou CTS). Lua é uma linguagem de script flexível e de sintaxe simples; linguagens de script são frequentemente usadas para juntar componentes escritos em outras linguagens, para construir protótipos de aplicações, e em arquivos de configuração.

Este trabalho apresenta duas abordagens de integração entre a linguagem Lua e o CLR, com o objetivo de permitir que scripts Lua instanciem e usem componentes escritos para o CLR. A primeira abordagem é a de criar uma ponte entre o interpretador Lua e o CLR, sem modificar o interpretador. Os recursos e a implementação desta ponte são mostrados, e ela é comparada com trabalhos que seguem a mesma abordagem.

A segunda abordagem é a de compilar as instruções da máquina virtual do interpretador Lua para instruções da Common Intermediate Language Do CLR, sem introduzir mudanças na linguagem Lua. A implementação de um compilador de instruções Lua para CIL é mostrada, e o desempenho de scripts compilados por ele é comparado com o desempenho dos mesmos scripts executados pelo interpretador Lua e com o de scripts equivalentes compilados por outros compiladores de linguagens de script para o CLR.

Palavras–chave

linguagem Lua; compiladores; Common Language Runtime; máquinas virtuais; Microsoft .NET; linguagens de script

PUC-Rio - Certificação Digital Nº 0210489/CA

Abstract

Queiroz, Fabio Mascarenhas de; Ierusalimschy, Roberto. **Integrating the Lua Language and the Common Language Runtime**. Rio de Janeiro, 2004. 68p. MSc. Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The Common Language Runtime (CLR) is a platform that aims to make the interoperability among different programming languages easier, by using a common language (the Common Intermediate Language, or CIL) and a common type system (the Common Type System, or CTS). Lua is a flexible scripting language with a simple syntax; scripting languages are frequently used to join components written in other languages, to build application prototypes, and in configuration files.
This work presents two approachs for integratiion between the Lua language and the CLR, with the objective of allowing Lua scripts to instantiate and use components written for the CLR. The first approach is to create a bridge between the Lua interpreter and the CLR, without changing the interpreter. The features and implementation of this bridge are shown, and it is compared with other work following the same approach.
The second approach is to compile the virtual-machine instructions of the Lua interpreter to instructions of the CLR's Common Intermediate Language, without introducing changes to the Lua language. The implementation of a Lua instructions to CIL compiler is shown, and the performance of scripts compiled by it is compared with the performance of the same scripts run by the Lua interpreter and with the performance of equivalent scripts compiled by compilers of other scripting language to the CLR.

Keywords

Lua language; compilers; Common Language Runtime; virtual machines; Microsoft .NET; scripting languages

PUC-Rio - Certificação Digital Nº 0210489/CA

Conteúdo

1 Introdução **11**
1.1 A Linguagem Lua 13
1.2 O Common Language Runtime 15

2 Interface entre o CLR e o Interpretador Lua **18**
2.1 Recursos da Interface 18
2.2 Implementação da Interface 27

3 Compilando Bytecodes Lua para CIL **34**
3.1 A Máquina Virtual de Lua 5.0 34
3.2 Mapeamento dos Tipos de Lua para o CTS 35
3.3 Compilação das Instruções 37
3.4 Implementando Outros Recursos de Lua 5.0 39
3.5 Interface com o CLR 42

4 Trabalhos Correlatos **44**
4.1 Pontes entre Lua e Outras Plataformas 44
4.2 Pontes entre Interpretadores e o CLR 46
4.3 Compiladores de Linguagens Dinâmicas para o CLR 49

5 Conclusão **55**

Referências Bibliográficas **59**

A Instruções da Máquina Virtual de Lua 5.0 **63**

PUC-Rio - Certificação Digital Nº 0210489/CA

Lista de Figuras

4.1 Tempos de chamada para métodos com parâmetros `System.Int32` 48
4.2 Tempos de chamada para métodos com parâmetros de tipos de objeto 48
4.3 Comparação entre compiladores para o CLR 52
4.4 Tempos de chamada para métodos 53

PUC-Rio - Certificação Digital Nº 0210489/CA

Lista de Tabelas

1.1 Tipos do CTS 16
1.2 Tipos primitivos 17

4.1 Scripts para o teste de desempenho dos compiladores 51

PUC-Rio - Certificação Digital Nº 0210489/CA

PUC-Rio - Certificação Digital Nº 0210489/CA

Realizing this [programming as sketching] has real implications for software design. It means that a programming language should, above all, be malleable. A programming language is for thinking of programs, not for expressing programs you've already thought of. It should be a pencil, not a pen. Static typing would be a fine idea if people actually did write programs the way they taught me to in college. But that's not how any of the hackers I know write programs. We need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types balanced on your knee and make polite conversation with a strict old aunt of a compiler.

Paul Graham, *Hackers and Painters.*

1
Introdução

O Common Language Runtime (CLR) é uma plataforma criada com o objetivo de facilitar a interoperabilidade entre diferentes linguagens de programação, através de uma linguagem intermediária (a Common Intermediate Language, ou CIL) e um sistema de tipos comum (o Common Type System, ou CTS) [12, 13]. A especificação do CLR é um padrão da International Standards Organization (ISO) e da Ecma International [14], e implementações existem para os sistemas operacionais Microsoft Windows, Linux, FreeBSD e MacOS X [15, 17]. Diversas linguagens têm compiladores para o CLR já prontos ou em desenvolvimento; uma relação destas linguagens pode ser encontrada na Internet [18].

A Common Language Specification (CLS) é um subconjunto da especificação do CLR. A CLS define as regras que os compiladores devem seguir para garantir a interoperabilidade entre as linguagens [14, CLI Partição I Seção 7.2.2]. Os compiladores que geram código capaz de usar bibliotecas que seguem as regras da CLS são chamados *consumidores CLS*. Os que podem produzir novas bibliotecas, ou estender bibliotecas existentes, são chamados *extensores CLS*.

O código gerado por um consumidor da CLS deve poder chamar qualquer método ou *delegate* que siga as regras da CLS, mesmo métodos com nomes inválidos na linguagem; chamar métodos com a mesma assinatura mas de interfaces diferentes; instanciar qualquer tipo que siga as regras da CLS, inclusive tipos aninhados; ler e atribuir a qualquer propriedade que siga as regras do CLS; registrar e remover tratadores para qualquer evento que siga as regras do CLS.

Linguagens de script são frequentemente usadas para conectar componentes escritos em outras linguagens. Também são usadas na construção de protótipos, e em arquivos de configuração. A natureza dinâmica das linguagens de script permite o uso de componentes sem a declaração prévia de tipos, e sem a necessidade de compilação. Ainda assim, elas checam a corretude de todas as operações em tempo de execução, fornecendo informações

PUC-Rio - Certificação Digital Nº 0210489/CA

detalhadas em caso de erros. Os recursos das linguagens de script podem aumentar a produtividade dos programadores em um fator de dois ou mais [3].

Lua [19, 1, 2] é uma linguagem de script flexível e com um interpretador pequeno, eficiente e fácil de embutir em outros programas. Lua tem uma sintaxe simples, é interpretada, dinamicamente tipada, e possui recursos reflexivos. Lua também possui funções como valores de primeira classe, escopo léxico, e co-rotinas.

Esta dissertação apresenta a integração entre a linguagem Lua e o CLR. O objetivo da integração foi oferecer todos os recursos de um consumidor completo da CLS, ou seja, permitir que scripts Lua instanciem e usem objetos do CLR, seguindo a sintaxe de Lua. Uma interface entre Lua e o CLR dá acesso a componentes escritos em qualquer linguagem que possa definir tipos para o CLR para os scripts Lua.

São usadas duas abordagens para obter a integração desejada. A primeira abordagem consistiu na criação de uma ponte entre o interpretador Lua e o CLR. O objetivo foi usar os recursos reflexivos da linguagem Lua e do CLR para construir esta ponte sem precisar fazer qualquer mudança no interpretador Lua. Sua operação não exige que os scripts sejam pré-processados ou que sejam criados *stubs* para os tipos do CLR que os scripts usam.

A segunda abordagem consistiu na compilação dos bytecodes da máquina virtual Lua para instruções da CIL. O objetivo foi construir um compilador de Lua para o CLR que oferecesse todos os recursos da linguagem, sem introduzir modificações na sua sintaxe ou no seu comportamento. A preocupação com o desempenho também esteve presente, em ambas as abordagens; seus desempenhos foram avaliados e comparados com os desempenhos de trabalhos similares.

A razão de se implementar a primeira abordagem é verificar se a integração entre Lua e o CLR é possível sem precisar de um compilador Lua, e definir como será a interface entre Lua e o CLR. O sucesso dessa abordagem mostra um caminho que outras linguagens de script podem usar para uma integração fácil com o CLR. Além da facilidade de implementação, em relação a um compilador, outra vantagem de uma ponte é a possibilidade de usar as bibliotecas já existentes para a linguagem Lua, em geral escritas em C. Isso também vale para outras linguagens de script, que têm bibliotecas escritas em C para melhor desempenho.

Implementar a segunda abordagem, por sua vez, permite verificar qual o grau de dificuldade de se implementar linguagens dinâmicas sobre o CLR, e quais empecilhos existem. A construção de compiladores de linguagens

PUC-Rio - Certificação Digital Nº 0210489/CA

de script para o CLR tem sido problemática. Os projetos de criação de compiladores das linguagens Perl e Python foram abandonados antes de serem concluídos [7, 8]. A companhia Smallscript Inc. vem trabalhando em um compilador da linguagem Smalltalk desde 1999, e nenhuma versão está publicamente disponível [9]. Um fator comum entre esses projetos é a sua ênfase em estender o CLR com a criação de novas classes *durante a compilação*. Essa ênfase dificulta a construção dos compiladores, pois criação e extensão dinâmica de tipos são recursos comuns das linguagens de script. A segunda abordagem deste trabalho pretende contornar as dificuldades encontradas na construção dos outros compiladores, enfatizando primeiro a implementação dos recursos da linguagem e segundo a utilização de tipos do CLR já existentes, abandonando a criação de novas classes durante a compilação.

O restante deste capítulo apresenta mais detalhadamente a linguagem Lua e o Common Language Runtime. O Capítulo 2 apresenta a primeira abordagem de integração e sua implementação, a biblioteca LuaInterface. O Capítulo 3 apresenta a segunda abordagem de integração e o compilador Lua2IL. O Capítulo 4 descreve outros trabalhos similares e uma avaliação do desempenho de ambas as abordagens. Finalmente, o Capítulo 5 resume as conclusões deste trabalho e sugestões para melhorias e trabalhos futuros.

1.1 A Linguagem Lua

Lua é uma linguagem de script projetada para ser facilmente embutida em aplicações, e usada como linguagem de configuração e extensão. Lua é usada atualmente em projetos no mundo todo, em especial jogos de computador; uma lista de alguns projetos que usam Lua está na Internet [24]. Lua tem uma sintaxe simples, e combina características de linguagens procedurais (estruturas de repetição e testes, atribuição, definição de funções com parâmetros e variáveis locais) com recursos mais avançados como arrays associativos, funções como valores de primeira classe, escopo léxico e co-rotinas.

O interpretador de Lua é implementado em ANSI C, como uma biblioteca. Os scripts Lua são primeiro compilados para uma representação intermediária (*bytecodes*) e em seguida executada por uma máquina virtual. A Seção 3.1 descreve a máquina virtual de Lua em mais detalhes. A biblioteca possui uma API que oferece funções para iniciar e controlar instâncias do interpretador, descrita mais detalhadamente na Seção 2.2.1.

PUC-Rio - Certificação Digital Nº 0210489/CA

PUC-Rio - Certificação Digital Nº 0210489/CA

Lua é uma linguagem dinamicamente tipada, e possui sete tipos: `nil`, *number*, *string*, *boolean*, *table*, *function* e *userdata*. O tipo `nil` tem apenas um valor, `nil`, e representa uma referência não inicializada (ou inválida); valores do tipo *number* são números de ponto flutuante com precisão dupla (o tipo `double` da linguagem C); valores do tipo *string* são cadeias de caracteres (o tipo `char` da linguagem C); o tipo *boolean* tem dois valores, `true` e `false`; valores do tipo *table* são tabelas, ou arrays associativos; funções pertencem ao tipo *function*; e o tipo *userdata* é para dados arbitrários da aplicação que está embutindo o interpretador.

As tabelas podem ser indexadas por qualquer valor, de qualquer tipo; o interpretador otimiza o acesso a índices inteiros, usando um vetor, e uma tabela hash armazena os valores para outros tipos de índices. Lua também oferece "açúcar sintático" para usar tabelas como registros (ou objetos): a expressão `tab.campo` é equivalente a `tab["campo"]`.

Funções são valores de primeira classe, e portanto podem ser passadas como argumentos para outras funções, atribuídas a variáveis, guardadas em tabelas e retornadas por funções. Lua também permite definir funções anônimas. Tabelas e funções de primeira classe permitem a criação de objetos em Lua; as funções armazenadas em uma tabela são os seus métodos. Lua também oferece açúcar sintático para definir métodos. Se `tab` é uma tabela, a declaração

```
function tab:metodo(param1,param2)
  -- código do método
end
```

é equivalente a

```
tab["metodo"]=function (self,param1,param2)
  -- código do método
end
```

e uma expressão como `tab:metodo(arg1,arg2)` é equivalente a `tab["metodo"](tab,arg1,arg2)`, ou seja, o objeto de destino da chamada vira o primeiro argumento da função.

Funções podem ser aninhadas arbitrariamente (o próprio corpo do script é uma função anônima), e as funções internas têm livre acesso às variáveis locais das funções externas (escopo léxico). Por exemplo, o trecho de código

```
function faz_contador()
  local cont=0
```

```
    return function ()
             cont=cont+1
             return cont
           end
  end
```

define uma função faz_contador que, a cada vez que é chamada, retorna uma função que incrementa e retorna o valor da variável cont. Como essa variável é local a uma função externa (a função faz_contador) o valor é "lembrado" entre diferentes invocações: a primeira chamada à função retornada por faz_contador retorna 1; a segunda chamada à mesma função retorna 2, e assim por diante.

A linguagem Lua também permite a extensão e modificação do comportamento de tabelas e userdata, em operações como indexação e comparação. As Seções 2.2.1 e 2.2.5 descrevem este recurso e como ele pode ser utilizado para estender o comportamento do interpretador.

Finalmente, as co-rotinas são linhas independentes de execução dentro de um script Lua, mas sem um escalonador; uma co-rotina tem que suspender explicitamente sua execução chamando uma função. Apenas uma co-rotina é executada por vez, mesmo que a máquina seja capaz de execução paralela. A Seção 3.4.1 mostra como as co-rotinas são criadas e usadas.

1.2
O Common Language Runtime

Como já dito no início deste capítulo, o CLR é uma plataforma criada para facilitar a interoperabilidade entre diferentes linguagens de programação. Ele oferece uma linguagem intermediária e um ambiente de execução comuns, com recursos como um *heap* gerenciado por um coletor de lixo, suporte a criação e sincronização de linhas de execução concorrentes (*threads*), segurança e autenticação de código. Além disso o CLR oferece um sistema de tipos comum, com recursos como reflexão e incorporação de meta-dados aos tipos. Uma extensa biblioteca de classes fornece uma API para os recursos da plataforma.

Os tipos do Common Type System (CTS), o sistema de tipos do CLR, se dividem em duas categorias: os *value-types*, ou valores, e os *reference types*, ou referências. A atribuição entre referências cria outra referência para os mesmos dados apontados pela referência que foi copiada; a atribuição entre valores cria uma cópia distinta do valor original. A Tabela 1.1 resume os tipos do CTS.

PUC-Rio - Certificação Digital Nº 0210489/CA

Valores	Referências
tipos primitivos	classes
estruturas	interfaces
enumerações	arrays
	delegates
	ponteiros

Tabela 1.1: Tipos do CTS

As referências vêm em diversos tipos, o principal deles sendo as *classes*. O CLR oferece herança simples para as classes, formando uma hierarquia que começa pela classe `System.Object`. Os objetos são instâncias das classes, alocadas no heap. As classes do CTS podem definir campos, propriedades, métodos e eventos. Propriedades são definidas por um par de métodos, um método para ler e outro para atribuir à propriedade, mas sintaticamente elas são como campos. Eventos são uma maneira de se registrar *callbacks* com um objeto, e são definidos por um par de métodos, um para adicionar e outro para remover um callback, e o tipo do callback (sua assinatura).

Outro tipo de referência são os *arrays*, vetores ou matrizes alocados no heap que contêm valores (ou referências) de um determinado tipo. As *interfaces* são tipos que só podem definir métodos e propriedades abstratos; outras classes podem implementar interfaces, e as interfaces podem herdar de outras interfaces. Os *delegates* são tipos que definem uma assinatura de um método; suas instâncias encapsulam métodos com essa assinatura, que podem ser chamados e combinados. Finalmente, os *ponteiros* são usados para passagem de valores por referência, em chamadas a métodos; eles apontam para um valor no heap ou na pilha de execução.

Os valores do CTS podem pertencer aos tipos primitivos, os escalares, ou a tipos definidos pelo usuário, as enumerações e estruturas. Os tipos primitivos do CLR estão resumidos na Tabela 1.2. Os tipos marcados com um asterisco (*) também fazem parte da Common Language Specification.

As enumerações são uma maneira de se associar nomes a valores inteiros, ou a bits em uma máscara de bits. As estruturas são um tipo composto, como as classes, podendo inclusive ter métodos e propriedades, mas que seguem a semântica de atribuição dos valores (e não possuem herança). A cada tipo de valor corresponde também um tipo de referência, usado para *boxing*, ou empacotamento dos valores no heap.

O sistema de execução do CLR é uma máquina virtual baseada em pilha, com cerca de 220 instruções (a Common Intermediate Language, ou

PUC-Rio - Certificação Digital Nº 0210489/CA

Nome do Tipo	Valores
System.Boolean*	Booleanos (`true` ou `false`)
System.Byte*	Naturais de 8 bits
System.SByte	Inteiros de 8 bits
System.Char*	Caracter unicode de 16 bits
System.Double*	Ponto flutuante com precisão dupla
System.Single*	Ponto flutuante com precisão simples
System.Int32*	Inteiros de 32 bits
System.UInt32	Naturais de 32 bits
System.Int64*	Inteiros de 64 bits
System.UInt64	Naturais de 64 bits
System.Int16*	Inteiros de 16 bits
System.UInt16	Naturais de 16 bits

Tabela 1.2: Tipos primitivos

CIL). A unidade básica de execução são os métodos; cada método possui um registro de ativação, mantido na pilha de execução do CLR. O registro de ativação contém um vetor com as variáveis locais do método, um vetor com seus argumentos, informação sobre o método em execução, uma referência para o registro do método que chamou o método atual, e a pilha de avaliação do método. As instruções da CIL cobrem operações como transferência de valores de e para a pilha de avaliação, criação de valores e referências, chamadas a métodos, operações aritméticas, operações em vetores, etc.

PUC-Rio - Certificação Digital Nº 0210489/CA

2
Interface entre o CLR e o Interpretador Lua

Este capítulo descreve a abordagem de integração entre a linguagem Lua e o Common Language Runtime, através da implementação de uma ponte, a biblioteca LuaInterface, entre o interpretador Lua 5.0 e o CLR.

LuaInterface usa a API do interpretador Lua e as APIs PInvoke (para interface com código nativo) e de reflexão do CLR. LuaInterface oferece as capacidades de um consumidor completo da Common Language Specification, ou seja, os scripts Lua podem instanciar e usar objetos do CLR, além de criar *delegates* a partir de funções Lua. A sintaxe para usar os objetos do CLR é a mesma sintaxe para usar objetos Lua. LuaInterface também fornece capacidade limitada de criar novas classes para o CTS.

Para as aplicações do CLR, LuaInterface oferece a possibilidade de executar código Lua, ler e escrever variáveis globais do interpretador, chamar funções Lua e registrar como funções os métodos de objetos CLR.

As seções seguintes descrevem em detalhes os recursos que LuaInterface disponibiliza e a sua implementação.

2.1
Recursos da Interface

LuaInterface encapsula a API da linguagem Lua em uma classe chamada `Lua`, que oferece métodos para executar código Lua, ler e escrever variáveis globais, e registrar métodos do CLR como funções Lua. Classes auxiliares oferecem métodos para acesso aos campos de tabelas Lua e para chamada a funções Lua.

Instanciar a classe `Lua` inicia um interpretador Lua. Várias instâncias podem ser criadas, e são independentes entre si. Os métodos `DoFile` e `DoString` da classe `Lua` executam respectivamente um arquivo fonte e um trecho de código Lua. Aplicações podem ler ou escrever variáveis globais indexando uma instância da classe `Lua` pelo nome da variável. O código C# seguinte demonstra o uso da classe `Lua`:

PUC-Rio - Certificação Digital Nº 0210489/CA

```
// Iniciar uma instância do interpretador Lua
Lua lua = new Lua();
// Executar trechos de código Lua
lua.DoString("num = 2"); // Cria variável global 'num'
lua.DoString("str = 'uma string'");
// Lê variáveis globais 'num' e 'str'
double num = (double)lua["num"];
string str = (string)lua["str"];
// Escreve na variável global 'str'
lua["str"] = "outra string";
```

2.1.1
Conversões de Tipos

O CLR é estaticamente tipado; logo, sempre que um valor Lua é passado para o CLR, LuaInterface converte o valor Lua para o tipo esperado pelo CLR, se possível; senão LuaInterface gera uma exceção. Se o CLR espera um valor de tipo `object`, LuaInterface usa o seguinte mapeamento: `nil` para `null`, números para `System.Double`, strings para `System.String` e booleanos para `System.Boolean`.

LuaInterface converte tabelas Lua para instâncias de `LuaTable`. Indexar uma instância desta classe acessa o campo da tabela correspondente à chave usada. Funções são convertidas para instâncias de `LuaFunction`. Esta classe define um método `call` que chama a função Lua correspondente e retorna um vetor com os valores de retorno da função.

Se o CLR espera um tipo numérico então LuaInterface converte números para o tipo esperado, arredondando o número, se preciso. LuaInterface também converte strings numéricas para números do CLR. Da mesma forma, LuaInterface converte números em strings, caso o CLR espere uma instância de `System.String`. Se o CLR espera um valor `System.Boolean` então LuaInterface converte qualquer valor Lua, exceto `false` e `nil`, para `true`.

Se o CLR espera um *delegate*, e o valor Lua é uma função, então LuaInterface converte a função para um delegate do tipo esperado. LuaInterface cria um objeto delegate que chama a função Lua, com os argumentos para o delegate se tornando argumentos para a função.

Sempre que um valor é passado do CLR para Lua, LuaInterface converte o valor CLR para o tipo Lua mais próximo: `null` para `nil`, valores numéricos (`System.Int32`, `System.Double`, etc.) para números, instâncias

PUC-Rio - Certificação Digital Nº 0210489/CA

de `System.String` para strings, instâncias de `System.Boolean` para booleanos. LuaInterface converte instâncias de `LuaTable` e `LuaFunction` para as tabelas e funções correspondentes, respectivamente.

LuaInterface converte todos os outros valores do CLR para *proxies* para o próprio valor. A Seção 2.1.4 cobre esta conversão.

2.1.2 Métodos, Construtores e Sobrecarga

O CLR permite sobrecarga de métodos. Sempre que um script Lua chama um método sobrecarregado, LuaInterface tem que escolher qual método deve chamar. LuaInterface passa por cada método, primeiro verificando se o número de parâmetros é igual ao número de argumentos da chamada. Se é igual então LuaInterface checa se cada argumento pode ser convertido para o tipo do seu parâmetro correspondente, de acordo com as regras descritas na Seção 2.1.1. LuaInterface gera um erro se não acha um método adequado para chamar. Construtores também podem ser sobrecarregados (e geralmente são). LuaInterface usa este mesmo procedimento para escolher qual construtor usar para instanciar um objeto.

Uma consequência deste procedimento de seleção é que podem haver métodos sobrecarregados que não podem ser chamados diretamente. Se o primeiro método recebe um parâmetro `System.Double`, e o segundo um parâmetro `System.Int32`, o segundo método nunca é escolhido; qualquer valor que pode ser convertido para `Int32` também pode ser convertido para `Double`. A Seção 2.1.6 mostra uma maneira de contornar esta situação.

2.1.3 Importando Tipos do CTS e Instanciando Objetos

Para instanciar novos objetos, os scripts Lua precisam de referências para os seus tipos. LuaInterface fornece duas funções para obter estas referências. A função `load_assembly` carrega uma assembly (biblioteca) do CLR e disponibiliza os seus tipos para importação pela função `import_type`. Esta função vasculha as assemblies carregadas em busca do tipo pedido e retorna uma referência para ele. O trecho de código Lua a seguir demonstra o uso das duas funções:

```
load_assembly("System.Windows.Forms")
load_assembly("System.Drawing")
Form = import_type("System.Windows.Forms.Form")
```

PUC-Rio - Certificação Digital Nº 0210489/CA

```
Button = import_type("System.Windows.Forms.Button")
Point = import_type("System.Drawing.Point")
StartPosition = import_type("System.Windows.Forms." ..
  "FormStartPosition")
```

A função `import_type` não funciona apenas para classes: scripts também podem obter referências para estruturas (ex. `Point`) e enumerações (ex. `FormStartPosition`) através dela.

Para instanciar um objeto do CLR, um script chama o seu respectivo tipo, como uma função. O exemplo seguinte estende o anterior para mostrar como objetos são instanciados:

```
form1 = Form()
botao1 = Button()
botao2 = Button()
posicao = Point(10,10)
posicao_inicio = StartPosition.CenterScreen
```

Se um tipo tem sobrecarga de seus construtores então LuaInterface escolhe qual construtor chamar usando o procedimento apresentado na Seção 2.1.2.

Uma vez importado o script pode usar um tipo para chamar seus métodos estáticos. A sintaxe é a mesma da chamada de métodos em objetos Lua; por exemplo, a expressão `Form:GetAutoScaleSize(arg)` chama o método `GetAutoScaleSize` do tipo `Form`. LuaInterface procura dinamicamente no tipo o método estático desejado.

Os scripts também podem usar um tipo para ler e escrever seus campos e propriedades estáticos. Por exemplo, o comando `var=Form.ActiveForm` atribui à variável `var` o valor da propriedade `ActiveForm` do tipo `Form`.

2.1.4 Acesso a Objetos do CTS

Alguns tipos do CTS têm um mapeamento direto para tipos básicos de Lua. Estes tipos são `null`, tipos numéricos, `System.Boolean`, `System.String` e os tipos `LuaTable` e `LuaFunction` apresentados na Seção 2.1.1.

LuaInterface mapeia todos os outros tipos do CTS para proxies para o respectivo valor. No exemplo de instanciação de objetos da Seção 2.1.3, por exemplo, os valores atribuídos às variáveis `form1`, `form2`, `botao1`, `botao2` e `posicao` são todos proxies para os objetos CLR reais.

PUC-Rio - Certificação Digital Nº 0210489/CA

Os scripts podem usar um proxy para um objeto CLR do mesmo jeito que usam qualquer outro objeto Lua: podem ler campos, atribuir valores a campos, ler propriedades, atribuir a propriedades, e chamar métodos. O próximo exemplo complementa os dois anteriores, mostrando como propriedades e métodos são usados:

```
botao1.Text = "Ok"
botao2.Text = "Cancelar"
botao1.Location = posicao
botao2.Location = Point(botao1.Left,
  botao1.Height + botao1.Top + 10)
form1.Controls:Add(botao1)
form1.Controls:Add(botao2)
form1.StartPosition = posicao_inicio
form1:ShowDialog()
```

No exemplo anterior, o comando `botao1.Texto="Ok"` atribui a string `"Ok"` à propriedade `Text` do objeto `botao1`. O comando `form1.Controls:Add(botao1)` lê a propriedade `Controls` do objeto `form1`, então chama o método `Add` do valor dessa propriedade, passando o objeto `botao1` como argumento para a chamada. Os três exemplos anteriores combinados, quando executados, exibem uma janela com dois botões no centro da tela.

LuaInterface também provê um atalho para indexar vetores, tanto para sua leitura quanto para sua escrita. O atalho é indexar a referência para o vetor com um número; por exemplo, `arr[3]`. Para matrizes, os scripts devem usar os métodos da classe `System.Array`.

LuaInterface passa para Lua, como um erro, qualquer exceção que ocorra durante a execução de um método do CLR. O objeto da exceção é a mensagem de erro (as “mensagens de erro” de Lua não precisam ser strings). O script podem usar os mecanismos de Lua para capturar e tratar estes erros.

2.1.5 Tratamento de Eventos

Eventos são um recurso do CLR para implementar *callbacks*. Para oferecer um evento, um tipo declara dois métodos: um método para adicionar um tratador para o evento e outro para remover um tratador. Os meta-dados para o tipo declaram o nome do evento, o tipo dos seus tratadores, e quais

PUC-Rio - Certificação Digital Nº 0210489/CA

métodos são usados para adicionar e remover tratadores. Uma aplicação pode usar a API de reflexão do CLR para descobrir quais eventos um tipo oferece, e para adicionar e remover tratadores de eventos; a API de reflexão obtém esta informação dos meta-dados.

LuaInterface representa eventos como objetos que definem dois métodos: `Add` e `Remove`. Estes métodos respectivamente adicionam e removem um tratador para o evento. O método `Add` recebe uma função Lua, converte a função para um delegate do tipo que o evento espera, e acrescenta o delegate como um tratador para aquele evento. O método `Remove`, em troca, recebe um delegate registrado como um tratador e o remove.

Por exemplo, se um objeto `obj` define um evento `Ev`, a expressão `obj.Ev` retorna um objeto que representa o evento `Ev`. Se `func` é uma função, o comando `obj.Ev:Add(func)` a registra como tratador do evento `Ev`. Toda vez que o evento `Ev` dispara o CLR chama o delegate e este em seguida chama a função `func`.

O trecho a seguir estende o exemplo da janela com dois botões, acrescentando tratadores de evento para os botões:

```
function trata_mouseup(sender,args)
  print(sender:ToString() .. " MouseUp!")
end
botao1.MouseUp:Add(trata_mouseup)
botao2.Click:Add(os.exit)
```

Neste exemplo, o comando `botao1.MouseUp:Add(trata_mouseup)` registra a função `trata_mouseup` como um tratador para o evento `MouseUp` do objeto `botao1`. Esta função escreve uma mensagem no console. O CLR passa os parâmetros `sender` e `args`; eles são, respectivamente, o objeto que disparou o evento e os dados específicos para aquele evento. O comando `botao2.Click:Add(os.exit)` registra a função `os.exit`, da biblioteca padrão de Lua, como tratador para evento `Click` do objeto `botao2`. Esta função encerra o programa. Ela não possui parâmetros, mas isto não é problema: o interpretador Lua descartará os dois argumentos passados à função.

2.1.6
Recursos Adicionais de um Consumidor CLS

Os recursos descritos na Seção 2.1 cobrem a maior parte das capacidades de um consumidor completo da CLS. Os próximos parágrafos apresen-

PUC-Rio - Certificação Digital Nº 0210489/CA

tam os recursos que cobrem o restante das capacidades que um consumidor completo precisa oferecer.

O CLR oferece tanto passagem de parâmetros por valor quanto por referência. Parâmetros passados por referência são de dois tipos: parâmetros *out* são parâmetros apenas para saída (seus valores não podem ser lidos), e os parâmetros *ref* podem ser lidos e terem valores atribuídos a eles.

A linguagem Lua oferece apenas passagem de parâmetros por valor, mas suas funções não estão limitadas a um único valor de retorno; LuaInterface usa este recurso para dar suporte a parâmetros *out* e *ref* nos métodos do CLR. Os valores de retorno de parâmetros *out* e *ref* são retornados após o valor de retorno do método.

Se o script chama um método sobrecarregado então LuaInterface executa a primeira versão com a assinatura compatível com os argumentos passados, logo alguns métodos de um objeto podem nunca ser escolhidos (como discutido na Seção 2.1.2). Para chamar esses métodos, LuaInterface oferece a função `get_method_bysig`. Ela recebe um objeto, o nome do método e referências para os tipos de sua assinatura. Chamar `get_method_bysig` retorna uma função que, quando chamada, executa o método que corresponde à assinatura passada. O primeiro argumento para a chamada deve ser o seu objeto de destino. Os scripts também podem usar `get_method_bysig` para chamar métodos das classes numéricas e de string do CLR, e para chamar métodos estáticos de tipos do CTS. Construtores também podem ser sobrecarregados, então também existe a função `get_constructor_bysig`.

Um consumidor completo da CLS deve permitir que métodos com nomes inválidos na linguagem sejam chamados. Em Lua as construções `obj:metodo(...)` e `obj["metodo"](obj,...)` são equivalentes, logo para chamar um método com um nome inválido em Lua basta usar a segunda construção. Por exemplo, a expressão `obj:function(...)` não é válida em Lua, pois `function` é uma palavra reservada. O script deve usar a expressão equivalente `obj["function"](obj,...)`, que é válida.

Caso um objeto possua mais de um método com o mesmo nome e assinatura, mas implementando interfaces diferentes, os scripts podem prefixar o nome do método com o nome da interface (a notação `INomeInterface.NomeMetodo` é usada pela API de reflexão do CLR). Por exemplo, se `obj` tem um método chamado `foo`, definido pela interface `IFoo`, deve-se chamar o método com a expressão `obj["IFoo.foo"](obj,...)`.

Finalmente, referências para tipos aninhados também podem ser obtidas facilmente com `import_type`. Basta chamar a função com o nome do tipo aninhado seguindo o nome do tipo que o contém

PUC-Rio - Certificação Digital Nº 0210489/CA

e um sinal de adição. Novamente, esta notação é usada pela API de reflexão do CLR. Um exemplo do uso desta notação é o comando `import_type("TipoExterno+TipoAninhado")`, que importa o tipo `TipoAninhado` dentro do tipo `TipoExterno`.

2.1.7 Criando Novas Classes com Lua

LuaInterface oferece a função `make_object` para criação de novas classes. A função recebe um objeto Lua e uma interface do Common Type System. LuaInterface automaticamente cria uma nova classe que implementa essa interface. O construtor desta classe recebe um objeto Lua e o guarda. Os métodos da interface delegam sua execução para métodos do objeto Lua armazenado. Depois de criar a classe, LuaInterface a instancia, passando o objeto Lua para o construtor. A função `make_object` retorna o objeto instanciado.

Por exemplo, seja `IExemplo` uma interface definida pelo seguinte código C#:

```
public interface IExemplo {
  float Tarefa(float arg1, float arg2);
}
```

A interface `IExemplo` define um método `Tarefa` que recebe dois parâmetros `float` e retorna outro `float`. Agora, seja `tab` uma tabela Lua definida pelo seguinte código Lua:

```
tab = {
  mult = 2
}

function tab:Tarefa(arg1, arg2)
  return self.mult*arg1*arg2
end
```

A tabela `tab` também define um método `Tarefa`, que recebe dois argumentos, os multiplica, e então multiplica o resultado por um campo de `tab` chamado `mult`, retornando o resultado final.

Definimos então uma classe que usa instâncias de `IExemplo`, com o código C# seguinte:

PUC-Rio - Certificação Digital Nº 0210489/CA

```
public class TesteExemplo {
  public static void FazTarefa(IExemplo ex, float arg1,
      float arg2) {
    Console.WriteLine(ex.Tarefa(arg1, arg2));
  }
}
```

A classe `TesteExemplo` define um método estático `FazTarefa` que recebe uma instância de `IExemplo` e dois valores `float`, chamando o método `Tarefa` da instância e passando os dois valores `float`. O resultado é exibido no console. Para concluir o exemplo, seja o código Lua seguinte:

```
IExemplo = import_type("IExemplo")
TesteExemplo = import_type("TesteExemplo")

obj = make_object(tab, IExemplo)
TesteExemplo:FazTarefa(obj, 2, 3)
```

Neste código, as primeiras duas linhas importam referências para a interface `IExemplo` e a classe `TesteExemplo` definidas anteriormente. A chamada a `make_object` cria uma instância da interface `IExemplo` que delega seu método `Tarefa` para `tab`. A última linha chama o método `FazTarefa` da classe `TesteExemplo`, passando a instância que criada por `make_object` e os números 2 e 3. Dentro de `FazTarefa`, o método `Tarefa` desta instância é chamado com os números 2 e 3. Na sequência, o método `Tarefa` de `tab` é chamado, novamente com 2 e 3 como argumentos, e o resultado (12) retornado para `FazTarefa`, que o exibe no console.

Sempre que um objeto Lua é passado onde o CLR espera uma interface, LuaInterface automaticamente chama `make_object` com o objeto Lua e o tipo da interface, e passa o objeto retornado por `make_object` no lugar. No exemplo anterior, o último trecho de código Lua poderia ser escrito da seguinte maneira, com o mesmo resultado (exibindo “12” no console):

```
TesteExemplo = import_type("TesteExemplo")
TesteExemplo:FazTarefa(tab, 2, 3)
```

A função `make_object` na verdade pode receber qualquer classe, não apenas interfaces. Ela cria uma nova subclasse da classe. Veja o manual da biblioteca LuaInterface [10] para mais detalhes.

PUC-Rio - Certificação Digital Nº 0210489/CA

2.2 Implementação da Interface

A biblioteca LuaInterface foi implementada na linguagem C#, com um pequeno trecho (menos de 30 linhas de código) em C. A biblioteca depende do interpretador Lua versão 5.0, assumindo a existência de uma biblioteca de vínculo dinâmico (DLL) chamada `lua-5.0.dll` contendo a implementação da API de Lua, e de outra biblioteca chamada `lualib-5.0.dll` contendo a implementação da API de biblioteca de Lua. A implementação dos recursos de LuaInterface é descrita nas seções seguintes.

2.2.1 A API de Lua

A API de Lua[1] é um conjunto de funções C que um programa pode usar para instanciar e se comunicar com interpretadores Lua. Uma instância do interpretador é criada com a função `lua_open`:

```
lua_State *lua_open(void);
```

O ponteiro retornado aponta para uma estrutura opaca para o programa, servindo apenas para ser passado como argumento para todas as outras funções da API.

Passagem de Valores

A API define uma *pilha virtual* para a passagem de valores de e para o interpretador. Cada elemento da pilha corresponde a um valor Lua. A API define funções para testar o tipo de um valor da pilha, funções para consultar valores da pilha e funções para empilhar valores.

As funções para consultar valores booleanos, numéricos e strings retornam o valor que está na pilha com o tipo C correspondente, respectivamente `int`, `double` e `char*`. Analogamente, as funções que empilham estes tipos de valores recebem o valor com o tipo C correspondente.

Outros valores não podem ser consultados diretamente, mas pode-se obter uma referência para eles e, com essa referência, empilhar de volta o valor.

[1] Esta seção é um resumo da informação contida no Capítulo 3 da Referência da Linguagem Lua [20] que tem relevância direta para a implementação da interface.

PUC-Rio - Certificação Digital Nº 0210489/CA

Userdata

Userdata é um tipo de Lua usado para armazenar dados arbitrários por aplicações que incorporam o interpretador Lua. O comportamento de um userdata pode ser estendido para permitir operações como indexação, aritmética e comparação.

A API de Lua oferece uma função para criar um userdata, alocando uma área na memória para seus dados, e uma função para obter um ponteiro para a área de memória de um userdata que está na pilha.

Tabelas e Metatables

A API oferece uma função para criar uma nova tabela no topo da pilha, assim como funções para indexar tabelas. As funções de indexação também são usadas para acessar variáveis globais.

Metatables são o mecanismo de Lua para estender o comportamento de objetos (tabelas e userdata). Uma metatable é uma tabela na qual alguns campos com nomes especiais, chamados de meta-métodos, são usados quando certas operações são feitas com o objeto. A API possui funções para atribuir uma metatable a um objeto e obter a metatable atribuída ao objeto, caso exista.

Funções

Para chamar uma função um programa a empilha seguida de seus argumentos, na ordem de chamada, e usa uma das funções da API para chamada de funções. O interpretador põe os valores de retorno na pilha na ordem em que são retornados.

A API também oferece uma função para empilhar ponteiros de função C como funções. A função C empilhada deve ter um único parâmetro `lua_State*`. Os argumentos para ela são passados na pilha, e a função deve empilhar os valores de retorno.

2.2.2 Platform Invoke

Platform Invoke, ou PInvoke, é a funcionalidade do CLR para chamada de código nativo [14, CLI Partition II Section 14.5.2]. PInvoke faz a transição entre o código do CLR e código nativo, fazendo também a conversão entre os tipos do CLR e os nativos. A conversão é dependente de plataforma, mas

PUC-Rio - Certificação Digital Nº 0210489/CA

toda implementação do CLR deve fazer a conversão bidirecional de pelo menos os tipos abaixo [14, CLI Partition II Section 14.5.4]:

- Tipos inteiros (inclusive `System.Boolean` e `System.Char`) para os seus equivalentes com a mesma largura;
- Enumerações, para o seu tipo de suporte;
- Tipos de ponto flutuante, para seus equivalentes com a mesma precisão;
- O tipo `System.String`, para vetores de caracteres;
- Ponteiros para quaisquer dos tipos acima, para instâncias de `System.IntPtr`, uma estrutura que encapsula ponteiros dentro do CLR.

As implementações do CLR devem fornecer conversões para o código nativo (mas não o contrário) dos seguintes tipos do CLR:

- Vetores de quaisquer dos tipos acima, para vetores do tipo correspondente;
- Delegates, para ponteiros de função.

Em C#, por exemplo, a definição PInvoke da função

```
void lua_pushstring(lua_State *L, const char* s);
```

seria, convertendo os tipos:

```
static extern void lua_pushstring(IntPtr L, string s);
```

O tipo `lua_State*`, nesse caso, é convertido como um ponteiro `void`, e fica totalmente opaco para o código C#.

PUC-Rio - Certificação Digital Nº 0210489/CA

PUC-Rio - Certificação Digital Nº 0210489/CA

2.2.3 Encapsulando a API

A tradução dos protótipos da API de Lua, em C, para assinaturas PInvoke, em C#, foi direta, pois os tipos envolvidos possuem um mapeamento automático via PInvoke. A exceção ficou por conta dos parâmetros de tipo ponteiro de função: embora PInvoke converta delegates para ponteiros de função automaticamente, ocorre um conflito entre as convenções de chamada no CLR da plataforma Microsoft .NET.

Os compiladores C usam a convenção CDECL[2] como padrão, logo Lua espera ponteiros de função que seguem esta convenção, mas o compilador Just-In-Time do CLR da Microsoft .NET usa a convenção STDCALL[3] como padrão. A solução foi escrever uma pequena extensão à API em C que encapsula o ponteiro para a função STDCALL dentro de uma função CDECL.

Com a API disponível para programas C#, a implementação da classe `Lua` foi simples. A API já tem funções para conversão de números, strings e valores booleanos de Lua para C e vice-versa. Como PInvoke, por sua vez, converte dos tipos C para os tipos do CTS, a classe `Lua` apenas chama as funções da API quando estes tipos estão envolvidos.

Para tabelas e funções, a API de Lua oferece funções para obter referências (números inteiros) para estes valores. A classe `Lua` guarda estas referências dentro de instâncias de `LuaTable` e `LuaFunction`. Os métodos destas classes obtêm o valor referenciado e chamam a função da API apropriada (ler ou escrever o valor de um campo ou chamar o valor como uma função).

2.2.4 Passando Objetos CLR para o Interpretador Lua

Para passar objetos CLR ao interpretador, LuaInterface usa um esquema similar ao usado para trazer tabelas e funções do interpretador Lua. O objeto primeiro é guardado em um vetor; um novo userdata é alocado, e a posição no vetor onde o objeto foi guardado é armazenada dentro do userdata. LuaInterface então guarda este userdata dentro de uma tabela Lua, usando como chave a posição do objeto CLR no vetor de objetos (o mesmo valor armazenado dentro do userdata). Dessa maneira

[2] A função chamadora limpa a pilha após a chamada.
[3] A função chamada limpa a pilha após a chamada.

o userdata é reaproveitado se o mesmo objeto CLR for passado novamente ao interpretador Lua, ao invés de se criar um novo.

Quando o userdata é coletado pelo interpretador Lua este procura um meta-método chamado `__gc` na metatable do userdata. LuaInterface define, para o userdata correspondente a um objeto CLR, um meta-método `__gc` que retira o objeto do vetor de objetos. A tabela Lua que guarda os userdata correspondentes a objetos CLR os guarda usando referências fracas[4], portanto ela não impede a coleta dos userdata.

2.2.5 Usando Objetos CLR de Lua

Em Lua, uma chamada a um método, como `obj:foo(arg1,arg2)`, equivale a indexar o objeto pelo nome do método e chamar o valor retornado passando o próprio objeto como primeiro argumento, seguido dos outros argumentos, ou seja, `obj["foo"](obj,arg1,arg2)`. Se `obj` for um userdata, a operação `obj["foo"]` faz o interpretador Lua procurar na metatable de `obj` por um meta-método chamado `__index`; o interpretador então chama o meta-método com o objeto e o nome do método como argumentos.

Para os objetos CLR, o meta-método `__index` procura no tipo do objeto, usando a API de reflexão do CLR, por um método com o nome passado ao meta-método. Se for encontrado, LuaInterface retorna um delegate representando este método. Se nenhum método for encontrado, `__index` retorna `nil`.

O interpretador Lua então chama o delegate retornado por `__index`, passando o objeto e os argumentos do método. O delegate consulta, na pilha de Lua, os argumentos para o método, faz a conversão dos argumentos para os tipos que o método espera, e chama o método. Se o método for sobrecarregado então o delegate primeiro verifica qual método é compatível com os argumentos passados, só então trazendo-os da pilha e chamando o primeiro método compatível.

Usar reflexão para procurar por um método no tipo de um objeto e depois criar um delegate para ele é custoso. Logo, LuaInterface define um cache para os delegates, para pagar o custo apenas uma vez, na primeira chamada ao método. LuaInterface mantém este cache nas metatables dos objetos CLR. Todos os objetos de um mesmo tipo compartilham a mesma metatable, e por consequência o mesmo cache. Basta um objeto de determinada classe chamar um método que todos os outros objetos da mesma classe

[4]Referências que não contam como tal para efeitos de coleta de lixo.

PUC-Rio - Certificação Digital Nº 0210489/CA

usarão o cache caso chamem o mesmo método, o que aumenta a eficiência do cache.

Se um método é sobrecarregado também existe o custo de procurar pelo método compatível com os argumentos passados. Mesmo sem sobrecarga existe o custo de decidir como os argumentos são convertidos para os tipos que o método espera. Logo, LuaInterface mantém um segundo cache dentro do delegate. O cache guarda o último método chamado e as funções usadas para converter os argumentos. O delegate primeiro tenta converter os argumentos e chamar o método usando a informação deste segundo cache; se ocorrer algum problema ele continua com o processo normal de procurar um método compatível com os argumentos.

Retornando ao exemplo do método `foo`, se `foo` for um campo de `obj` o meta-método `__index` encontra aquele campo no tipo de `obj`, usando reflexão, e retorna o valor do campo. O meta-método `__index` também guarda o campo em um cache, para pagar o custo da procura apenas na primeira leitura do campo. Se `foo` for uma propriedade ou um evento então a operação é feita de forma análoga. Se o campo ou a propriedade não existem LuaInterface retorna `nil`.

Quando o script tenta atribuir um valor ao campo, como em `obj.foo=val`, e `obj` é um userdata, Lua procura na metatable do userdata por um meta-método chamado `__newindex`, chamando o meta-método com o objeto, o nome do campo e o valor como argumentos. O meta-método `__newindex` procura no tipo do objeto por aquele campo, converte o valor para o seu tipo e atribui o valor convertido a ele. A atribuição a uma propriedade é feita de maneira análoga. O meta-método `__newindex` aproveita o cache do meta-método `__index`. Se o campo ou a propriedade não existem LuaInterface gera um erro.

Os tipos retornados pela função `import_type` são instâncias da classe `Type`; suas buscas reflexivas (para métodos, campos, etc.) são restritas a métodos estáticos, mas fora isso são como outras instâncias de objetos. Suas metatables também possuem um meta-método `__call`; Lua chama este meta-método quando o tipo o script chama o tipo como uma função. A implementação de `__call` procura entre os construtores do tipo por um compatível com os argumentos passados e instancia um objeto usando este construtor.

PUC-Rio - Certificação Digital Nº 0210489/CA

2.2.6
Delegates, Eventos e Interfaces

Sempre que uma função Lua é passada onde o CLR espera um delegate, LuaInterface cria dinamicamente uma subclasse de `LuaDelegate`. Esta subclasse define um método com a assinatura do delegate, e um construtor que recebe uma função Lua como argumento. LuaInterface então cria uma instância desta subclasse e cria o delegate a partir desta instância. O delegate criado é passado para o CLR.

A subclasse de `LuaDelegate` é gerada usando a API `Reflection.Emit` do CLR. A API `Reflection.Emit` tem classes para gerar *assemblies*, tipos, e emitir bytecodes da Common Intermediate Language. Os tipos criados pela API podem ser mantidos na memória (e serem temporários) ou gravados em disco (e serem permanentes). As subclasses de `LuaDelegate` que LuaInterface gera são mantidas apenas na memória. LuaInterface guarda a classe em um cache, e ela é reutilizada se LuaInterface precisar de outro delegate do mesmo tipo.

LuaInterface retorna eventos como objetos que implementam um método `Add` e um método `Remove`. O método `Add` recebe uma função Lua e cria um delegate com a mesma assinatura dos tratadores do evento. O método `Add` então registra este delegate como um tratador do evento, e retorna o delegate. O método `Remove` recebe um delegate previamente registrado por `Add` e o remove.

A função `make_object` também usa a API `Reflection.Emit` do CLR para gerar suas classes.

PUC-Rio - Certificação Digital Nº 0210489/CA

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.

PUC-Rio - Certificação Digital Nº 0210489/CA

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

PUC-Rio - Certificação Digital Nº 0210489/CA

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.

PUC-Rio - Certificação Digital Nº 0210489/CA

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

PUC-Rio - Certificação Digital Nº 0210489/CA

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.

PUC-Rio - Certificação Digital Nº 0210489/CA

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

PUC-Rio - Certificação Digital Nº 0210489/CA

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].

PUC-Rio - Certificação Digital Nº 0210489/CA

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 põe 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.

PUC-Rio - Certificação Digital Nº 0210489/CA

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

PUC-Rio - Certificação Digital Nº 0210489/CA

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.

PUC-Rio - Certificação Digital Nº 0210489/CA

4
Trabalhos Correlatos

Este capítulo apresenta trabalhos relacionados aos trabalhos descritos nos dois capítulos anteriores (LuaInterface e Lua2IL). Os trabalhos são divididos em três categorias: pontes entre o interpretador Lua e outras plataformas, pontes entre outros interpretadores e o CLR, e compiladores de outras linguagens dinâmicas para o CLR. Cada categoria é apresentada em uma seção.

As pontes são comparadas com LuaInterface, tanto em relação aos recursos oferecidos quanto à implementação do mesmos. Para as pontes com o CLR, é feita uma comparação de desempenho com LuaInterface. Os compiladores são comparados com Lua2IL, e o desempenho do código que eles geram é comparado com o desempenho do código compilado por Lua2IL.

PUC-Rio - Certificação Digital Nº 0210489/CA

4.1
Pontes entre Lua e Outras Plataformas

Nesta seção são apresentadas as pontes LuaORB e LuaJava, entre o interpretador Lua e as plataformas CORBA e Java, respectivamente. O objetivo da plataforma CORBA é a interoperabilidade entre componentes heterogêneos dentro de um ambiente distribuído, um objetivo similar ao do CLR de interoperabilidade entre linguagens de programação; LuaORB é a inspiração para o trabalho apresentado nesta dissertação. Já a plataforma Java tem recursos e implementação similares aos do CLR. Tanto LuaORB quanto LuaJava são comparadas com LuaInterface.

LuaORB

LuaORB é uma biblioteca para a linguagem Lua, implementada em C++, para controlar objetos e implementar interfaces CORBA [4, 11]. LuaORB usa os recursos reflexivos do padrão CORBA para criar proxies

para objetos CORBA em tempo de execução. Os scripts usam a sintaxe de Lua para acesso a propriedades e para chamada a métodos destes objetos. A conversão dos tipos de Lua para os tipos CORBA é automática; LuaORB também converte automaticamente tabelas Lua em estruturas CORBA. Scripts Lua também podem usar LuaORB para registrar tabelas Lua como implementações de interfaces CORBA. LuaORB registra os objetos dinamicamente, usando a CORBA Dynamic Skeleton Interface.

LuaInterface é similar a LuaORB em sua operação, oferecendo aos objetos CLR o mesmo nível de acesso que LuaORB oferece aos objetos CORBA. A exceção é a conversão automática de tabelas em estruturas, mas LuaInterface permite instanciar e usar estruturas do CLR. LuaInterface também oferece a possibilidade de usar tabelas Lua como objetos do CLR (a função `make_object`), criando dinamicamente as classes necessárias usando a API `Reflection.Emit` do CLR.

LuaJava

LuaJava é uma ponte entre Lua e a máquina virtual da linguagem Java, e permite que scripts Lua instanciem e usem objetos Java e que criem classes a partir de tabelas Lua [5, 6]. Para comunicação com o interpretador Lua, LuaJava usa a API de reflexão e a API de acesso a código nativo de Java. Quando um script instancia um objeto Java, LuaJava retorna um proxy para o objeto, e o script usa este proxy como um objeto Lua qualquer. LuaJava cria novas classes Java a partir de tabelas Lua por geração e carregamento dinâmico de bytecodes da máquina virtual Java.

LuaInterface é bastante similar em recursos a LuaJava, mas sua implementação é mais simples, devido às facilidades oferecidas pela CLR: as APIs PInvoke e `Reflection.Emit`. Para chamar funções da API de Lua a partir de Java, LuaJava possui *stubs* em C que fazem a conversão dos tipos Java para os tipos C. PInvoke faz as conversões automaticamente, bastando declarar as funções da API. Já `Reflection.Emit` oferece métodos que criam e carregam classes temporárias dinamicamente; Java não possui uma API similar, então LuaJava precisa criar e carregar os bytecodes que representam a classe diretamente.

PUC-Rio - Certificação Digital Nº 0210489/CA

4.2
Pontes entre Interpretadores e o CLR

Esta seção apresenta pontes publicamente disponíveis entre outras linguagens interpretadas e o CLR, comparando-as com LuaInterface. São apresentadas a LuaPlus, PerlNET e Dot-Scheme, pontes respectivamente entre o CLR e os interpretadores Lua, Perl e Scheme. A seção também apresenta uma comparação de desempenho entre LuaInterface e PerlNET.

LuaPlus

A distribuição LuaPlus é uma interface de C++ para o interpretador Lua que também inclui uma interface do CLR [22]. A interface CLR possui métodos para executar código Lua, ler e atribuir valores a variáveis globais de Lua, e registrar delegates como funções Lua. Objetos CLR são passados para o interpretador Lua como userdata, mas scripts Lua não podem usá-los (chamando seus métodos, por exemplo), nem podem instanciar novos objetos. Os delegates registrados como funções também não podem possuir qualquer assinatura, mas sim uma assinatura fixa, definida pela própria interface. LuaPlus, portanto, oferece apenas uma pequena parte dos recursos presentes em LuaInterface.

PerlNET

PerlNET é uma biblioteca comercial, desenvolvida pela ActiveState, para integrar o interpretador da linguagem Perl ao CLR [16]. Ela usa PInvoke para comunicação entre o interpretador Perl 5.6 e o CLR. Scripts Perl podem, com PerlNET, instanciar e usar objetos do CLR através da mesma sintaxe usada com objetos Perl. Também podem definir novas classes do CLR, com métodos implementados em Perl. As classes são permanentes, e podem definir novos métodos, visíveis a partir de outros objetos do CLR.

PerlNET e LuaInterface são equivalentes quanto ao uso de objetos CLR; ambos são consumidores completos da Common Language Specification. PerlNET oferece mais recursos quanto à extensão do CLR: as classes criadas por LuaInterface são temporárias, e podem apenas redefinir métodos de suas classes base. PerlNET empacota o interpretador Perl, o script que define a nova classe e a classe gerada em uma única *assembly* do CLR, e as aplicações do CLR podem referenciar esta assembly para criar instâncias da nova classe e chamar os seus métodos.

PUC-Rio - Certificação Digital Nº 0210489/CA

Dot-Scheme

Dot-Scheme é uma ponte entre o interpretador PLT Scheme e o CLR. Programas Scheme podem instanciar e usar objetos do CLR, com os métodos dos objetos mapeados para funções Scheme. Dot-Scheme não é um consumidor completo do CLS: não oferece nenhuma maneira de definir delegates em código Scheme, para tratar eventos, por exemplo. Também não há nenhum recurso de criação de novas classes. Oferece, portanto, menos recursos que LuaInterface.

4.2.1
Comparação de Desempenho

Esta seção apresenta uma avaliação do desempenho de LuaInterface, comparando-o com o desempenho da biblioteca PerlNET. A biblioteca LuaPlus não oferece o recurso necessário para essa comparação (chamadas a métodos de objetos CLR), e Dot-Scheme, de acordo com o seu autor, não está otimizada para melhor desempenho, logo estas duas pontes foram deixadas de fora da comparação. A Seção 4.3.1 compara o desempenho de LuaInterface com o do código compilado por Lua2IL.

O foco dos testes de desempenho foi nas chamadas de métodos do CLR a partir de scripts, já que este é o principal recurso oferecido pelas pontes. Ao todo foram feitas chamadas a seis métodos diferentes, variando o número e tipo dos parâmetros e do valor de retorno.

Três dos métodos têm todos os parâmetros e o valor de retorno do tipo `System.Int32`, e variam no número de parâmetros (zero, um ou dois). Suas assinaturas são, respectivamente, `Int32 ()`, `Int32 (Int32)` e `Int32 (Int32, Int32)`. Os tempos para as chamadas a estes métodos estão na Figura 4.1. Todos os tempos estão em microsegundos, e foram coletados na mesma máquina, sob as mesmas condições[1]. Os tempos são uma média de dez execuções distintas de um milhão de chamadas cada.

A coluna *MethodBase.Invoke* mostra os menores tempos possíveis para chamadas reflexivas aos métodos, ou seja, o tempo para a chamada uma vez que já se tenha o método e os argumentos necessários. A coluna *Cache* mostra o tempo para uma chamada a partir de Lua, quando o método

[1] Athlon 1.2GHz, com 256Mb de memória, sob o sistema operacional Windows XP Professional, e a versão 1.1 do .NET Common Language Runtime. O interpretador Lua foi compilado com o compilador Microsoft Visual C++ versão 7, com todas as otimizações ativas. A biblioteca PerlNET usada é a versão que acompanha o Perl Dev Kit 5.3, da ActiveState, e já estava compilada. Nenhuma outra aplicação estava executando durante a coleta dos tempos.

PUC-Rio - Certificação Digital Nº 0210489/CA

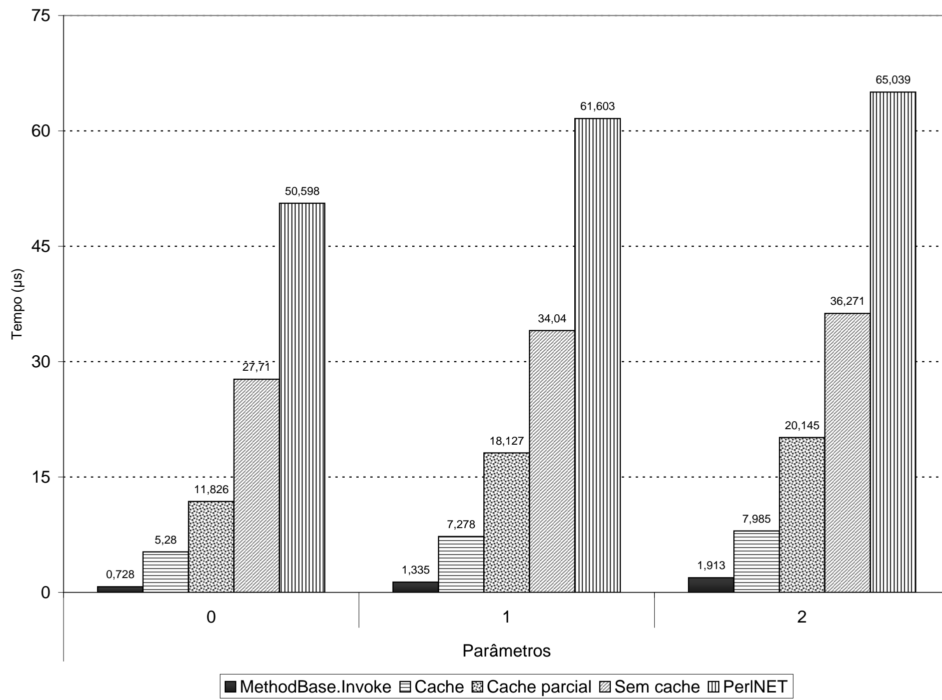


Figura 4.1: Tempos de chamada para métodos com parâmetros `System.Int32`

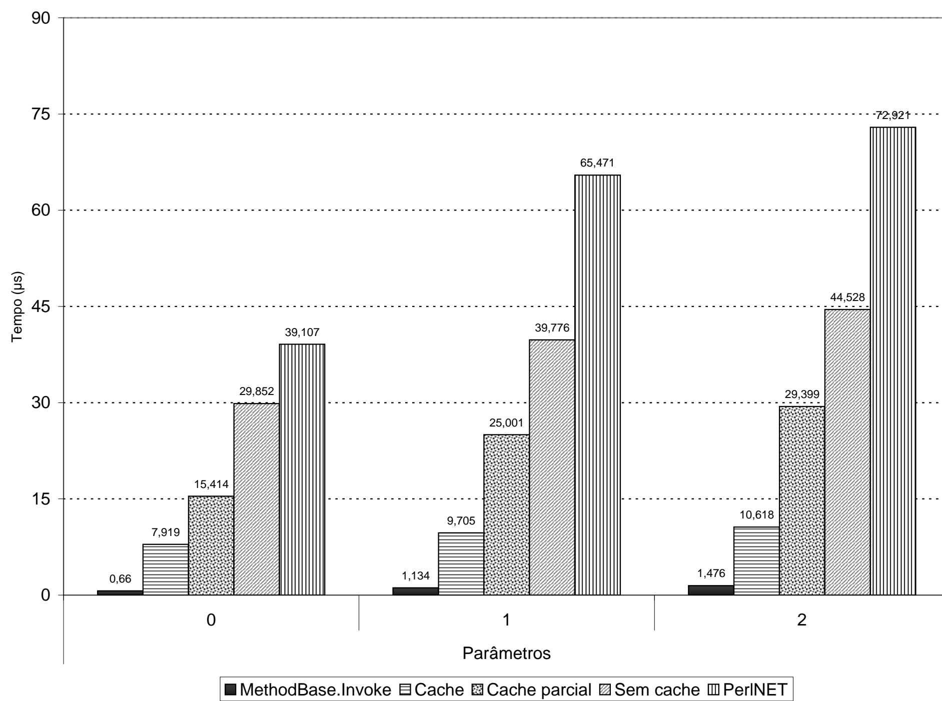


Figura 4.2: Tempos de chamada para métodos com parâmetros de tipos de objeto

PUC-Rio - Certificação Digital Nº 0210489/CA

já está em cache (a partir da segunda chamada). A coluna *Cache parcial* mostra o tempo com o cache interno do delegate desativado, forçando que LuaInterface, a cada chamada, descubra quais tipos o método recebe e como fazer a conversão dos valores Lua passados para esses tipos (também é o tempo para uma chamada em que ocorre incompatibilidade de tipos). Finalmente, a coluna *Sem cache* mostra o tempo com ambos os caches desativados, forçando LuaInterface a procurar pelo método a cada chamada (o tempo para a primeira chamada a um método). A coluna *PerlNET* mostra os tempos para chamadas a métodos a partir da linguagem Perl, usando a ponte PerlNET.

Os três métodos restantes para os quais foram feitos testes de desempenho têm parâmetros e valor de retorno de um tipo de objeto; o seu tipo é a própria classe na qual os métodos estão definidos. Os três métodos também variam na quantidade de parâmetros, de zero a dois, e suas assinaturas são `PerfTest ()`, `PerfTest (PerfTest)` e `PerfTest (PerfTest, Perftest)`, respectivamente. A Figura 4.2 mostra os tempos para estes três testes. Fica evidente, na diferença entre os tempos das chamadas a partir de Lua, o alto custo de se procurar reflexivamente o método a ser chamado, e o custo de determinar como cada argumento deve ser convertido. Os tempos com os caches ativos são de cerca de um quinto do tempo sem os caches.

O pequeno aumento nos tempos das chamadas a partir de Lua, da Figura 4.1 para a Figura 4.2, se deve aos testes que LuaInterface precisa fazer nos valores de tipos de objeto, antes de trazê-los para o CLR. Valores destes tipos são representados em Lua por userdata, e quando LuaInterface encontra um userdata ela checa primeiro se o userdata realmente corresponde a um objeto CLR.

Finalmente, a própria API PInvoke representa um custo. Cada chamada PInvoke gera de dez a trinta instruções, possivelmente mais, a depender do tipo dos argumentos envolvidos [23]. Individualmente as chamadas contam pouco, mas, como a chamada de um método envolve várias chamadas à API de Lua, os tempos individuais se somam até chegar a cerca de um quinto do tempo total da chamada ao método.

4.3 Compiladores de Linguagens Dinâmicas para o CLR

Esta seção compara alguns compiladores de outras linguagens dinâmicas para o CLR com o compilador Lua2IL. Como o único compilador totalmente implementado é o da linguagem JScript, mesmo compiladores

PUC-Rio - Certificação Digital Nº 0210489/CA

em desenvolvimento (ou abandonados) são apresentados. A seção também apresenta uma comparação de desempenho entre o código gerado pelos compiladores, e uma comparação do desempenho das chamadas a métodos CLR entre os compiladores e LuaInterface.

Python for .NET

Em 1999 e 2000 a Microsoft financiou uma pesquisa para a criação de um compilador Python para o CLR, o Python for .NET [8]. Python for .NET percorre a árvore sintática gerada pelo interpretador CPython, emitindo código da Common Intermediate Language do CLR através da API `Reflection.Emit`. A implementação tem similaridades com a de Lua2IL: Python for .NET define uma estrutura `PyObject` para seus valores, e uma interface IPyType que define as operações que podem ser feitas sobre os valores (os equivalentes de Lua2IL são a estrutura LuaValue e a classe LuaReference, respectivamente).

Cerca de 95% do núcleo da linguagem Python está disponível para os scripts, de acordo com o autor. Ficaram de fora do compilador os tipos primitivos sem um correspondente direto no CLR (números de tamanho arbitrário, números complexos e elipses), além de métodos embutidos das classes Python, usados para extensão dinâmica de classes e objetos. A sintaxe da linguagem não foi modificada. O desenvolvimento do compilador Python for .NET foi interrompido há cerca de dois anos. O último protótipo disponível tem data de abril de 2002, com partes datando de abril de 2000.

PUC-Rio - Certificação Digital Nº 0210489/CA

Perl for .NET

Perl for .NET é um compilador da linguagem Perl para o CLR, desenvolvido pela ActiveState entre 1999 e 2000 [7]. O compilador funciona como um *back-end* para o interpretador Perl 5.6, gerando código C# que chama um runtime Perl para as operações. Não há informações sobre o quanto da linguagem Perl é coberto pelo compilador, e o código-fonte para o mesmo não está disponível. A última versão data de junho de 2000, e funciona apenas com uma versão beta do CLR, não mais disponível.

JScript .NET

JScript .NET é uma extensão da linguagem JScript (ou EcmaScript) com um compilador para o CLR, parte do .NET Software Development

Script	Parâmetro
ack	Cálculo da função de Ackermann, parâmetros 3 e 8
fibo	Cálculo dos números de Fibonacci, o 30º número
random	Gerador de números aleatórios, gerar 1.000.000 de números entre 0 e 100
sieve	Crivo de Eratóstenes, de 2 até 8.192, 10 execuções
matrix	Multiplicação de matrizes $30x30$, 100 execuções
heapsort	Heapsort em um vetor de 10.000 números gerados aleatoriamente

Tabela 4.1: Scripts para o teste de desempenho dos compiladores

Kit da Microsoft [25]. O compilador estende a linguagem com classes e declarações opcionais de tipos, mas mantém os recursos dinâmicos de JScript. Para uso integral da interface do código JScript com o CLR, entretanto, são necessárias declarações de tipos; os scripts podem instanciar e usar classes do CLR sem declarações, mas delegates devem ser declarados com a assinatura correta, e dentro de uma classe. O código gerado pelo compilador usa os tipos do CLR nativamente, não os encapsulando dentro de outros valores. O resultado é que todas as operações envolvem checagem de tipos e *casts*.

S#.NET

S# é uma versão da linguagem Smalltalk desenvolvida pela SmallScript Corporation, e S#.NET é um compilador de S# para o CLR. Segundo o autor, o compilador e o runtime da linguagem estão prontos, mas ainda será feita a integração com o ambiente Visual Studio.NET. Não há uma versão pública para avaliação. O compilador vem sendo desenvolvido desde 1999.

4.3.1 Comparação de Desempenho

O primeiro teste de desempenho foi a execução de seis scripts do *The Great Win32 Computer Language Shootout* [26], envolvendo principalmente operações aritméticas, recursão e operações com vetores. O objetivo foi avaliar o desempenho do código gerado pelos compiladores para as operações simples das linguagens. A descrição de cada script de teste e os parâmetros de sua execução estão na Tabela 4.1.

PUC-Rio - Certificação Digital Nº 0210489/CA

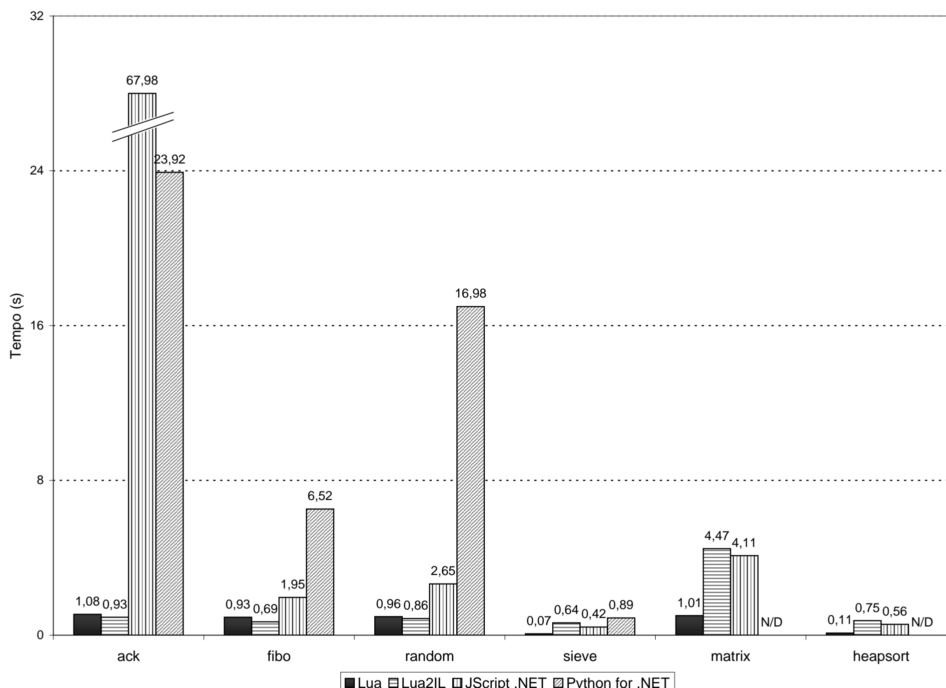


Figura 4.3: Comparação entre compiladores para o CLR

Foram testados os compiladores Lua2IL, JScript .NET e Python for .NET. Os mesmos scripts compilados pelo Lua2IL também foram executados pelo interpretador Lua 5.0. Os resultados são mostrados na Figura 4.3. Os tempos são em segundos, e todos os scripts foram executados na mesma máquina, sob as mesmas condições[2].

Os scripts *matrix* e *heapsort* não foram compilados pelo Python for .NET, mas não apresentavam erros de sintaxe (foram executados normalmente pelo interpretador Python).

Não é surpresa que o compilador Python for .NET tenha apresentado os piores tempos, já que seu desenvolvimento foi abandonado antes mesmo dele implementar toda a linguagem, e antes de se otimizar o código gerado por ele. O desempenho de JScript .NET nos três primeiros scripts também foi o esperado, por todas as operações envolverem casts e checagens de tipos. Já o pior desempenho de Lua2IL nos três últimos scripts é devido ao fato dos vetores serem implementados por tabelas hash, no protótipo atual de Lua2IL. As tabelas do interpretador Lua têm tanto uma parte vetor quanto uma parte hash, e índices numéricos usam a parte vetor. Uma otimização

[2]Athlon 1.2GHz, com 256Mb de memória, sob o sistema operacional Windows XP Professional. O código gerado pelos compiladores foi executado pela versão 1.1 do .NET Common Language Runtime. O interpretador Lua foi compilado com o compilador Microsoft Visual C++ versão 7, com todas as otimizações ativas. Nenhuma outra aplicação estava executando durante a coleta dos tempos.

PUC-Rio - Certificação Digital Nº 0210489/CA

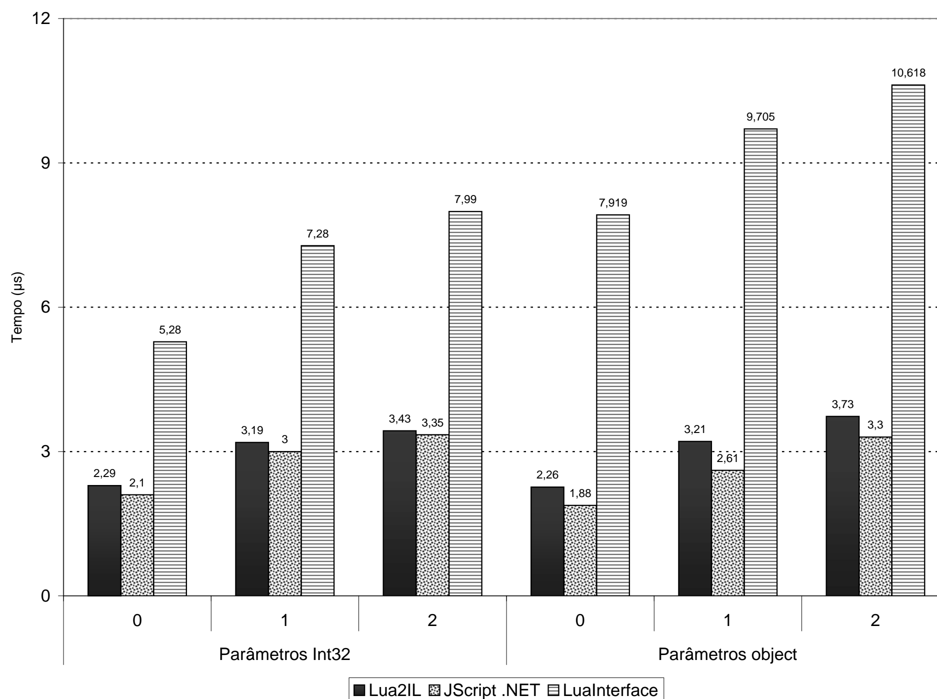


Figura 4.4: Tempos de chamada para métodos

PUC-Rio - Certificação Digital Nº 0210489/CA

similar, no código gerado por Lua2IL, deve reduzir bastante os tempos dos últimos três scripts.

O segundo teste de desempenho é uma extensão da comparação de desempenho da Seção 4.2.1, e avalia os tempos para chamadas a métodos de um objeto CLR, a partir do código gerado pelo compilador Lua2IL e pelo compilador JScript .NET (usando o recurso de *late binding* do compilador, ou seja, sem declarar tipos). O compilador Python for .NET ficou fora deste teste, pois os scripts compilados por ele só podem instanciar tipos da biblioteca padrão do CLR, e não puderam instanciar os tipos da assembly usada nos testes.

Os tempos são mostrados na Figura 4.4, e estão em microsegundos. Foram coletados na mesma máquina e sob as mesmas condições (as mesmas do teste anterior). As colunas *LuaInterface* são os tempos das chamadas a partir do interpretador Lua, usando a biblioteca LuaInterface, com o cache ativo (coluna *Cache* das Figura 4.1 e 4.2). As colunas *Lua2IL* são os tempos para chamar um método a partir do código gerado pelo compilador Lua2IL, também com o cache ativo. As colunas *JScript .NET* são os tempos para chamar um método a partir do código gerado pelo compilador JScript .NET.

Para este teste, o código gerado pelo compilador JScript .NET tem uma ligeira vantagem, por usar os valores do CLR diretamente, enquanto o código gerado por Lua2IL mapeia todos os valores seguindo as regras da

Seção 3.2; todos os valores passados para um método do CLR têm que ser mapeados de `LuaValue` para o valor do CLR correspondente, e o valor de retorno do método tem que ser mapeado para um valor `LuaValue`.

O teste também indica que a maior parte do tempo das chamadas a partir do interpretador Lua se deve aos custos de passar valores entre o interpretador Lua e o CLR, além dos custos das operações sobre valores userdata, envolvendo metatables; o código gerado por Lua2IL não precisa pagar nenhum destes custos.

PUC-Rio - Certificação Digital Nº 0210489/CA

5 Conclusão

Esta dissertação apresentou duas abordagens para integração entre a linguagem Lua e o Common Language Runtime. O objetivo principal da integração foi permitir que scripts Lua instanciem e usem objetos do CLR, com todos os recursos exigidos de um consumidor completo da Common Language Specification.

A primeira abordagem foi a criação de uma ponte entre o interpretador Lua e o CLR, com o objetivo de obter a integração entre os dois ambientes sem precisar modificar o interpretador e a linguagem Lua. O resultado foi a implementação da biblioteca LuaInterface, que cumpre os objetivos desejados. LuaInterface também permite que aplicações do CLR executem código Lua, e oferece recursos limitados para criação de classes do CLR, com métodos implementados por código Lua.

LuaInterface é implementada em C#, com uma parte mínima em C. A interface entre o código C# e o interpretador é através da API PInvoke, a interface de código nativo do CLR. O acesso ao interpretador Lua foi direto, pois ele foi criado para ser facilmente embutido em outras aplicações. Algumas classes em C# encapsulam a API de Lua, oferecendo para as aplicações CLR uma interface mais natural que as funções da API de Lua.

A principal dificuldade na implementação de LuaInterface se deu no suporte a eventos e nos recursos de criação de delegates e novas classes. Todos estes recursos exigiram geração dinâmica de código CIL, enquanto que todos os outros recursos puderam ser implementados em C#. O código gerado dinamicamente é difícil de depurar, portanto foi deixado o mais simples possível.

O desempenho da biblioteca foi avaliado, mais especificamente o tempo total de uma chamada a um método do CLR, a partir de um script Lua. LuaInterface guarda os métodos chamados em um cache, o que reduz o tempo da segunda chamada em diante para um quinto do tempo da primeira chamada a um método. Na comparação com uma biblioteca similar para a linguagem Perl, comercial, as chamadas a partir de LuaInterface (em cache)

PUC-Rio - Certificação Digital Nº 0210489/CA

levaram de um oitavo a um quinto do tempo das chamadas a partir de Perl.

Conclusões obtidas durante a implementação da biblioteca LuaInterface:

– O fato de Lua ser uma linguagem extensível facilitou a implementação de todos os recursos de um consumidor completo do CLS sem precisar de mudanças no interpretador ou na linguagem. O resultado é que, para o programador, as operações sobre objetos do CLR seguem a mesma sintaxe das operações sobre objetos da linguagem Lua.
– A API de reflexão do CLR também foi essencial para a abordagem implementada por LuaInterface, já que Lua é uma linguagem dinamicamente tipada; a API de reflexão permite carregar, instanciar e usar qualquer tipo do CLR em tempo de execução.
– As chamadas reflexivas não são o gargalo da biblioteca; as buscas reflexivas seriam um gargalo se não fosse a presença do cache de métodos, entretanto.
– A API PInvoke é muito fácil de se usar, mas mostrou-se responsável por cerca de um quinto do tempo das chamadas com cache, por isso foi preciso reduzir a comunicação entre o interpretador e o CLR ao mínimo possível.

A segunda abordagem de integração foi a compilação dos bytecodes da máquina virtual Lua para a Common Intermediate Language do CLR, com o objetivo de oferecer todos os recursos da linguagem Lua ao código compilado, e não introduzindo modificações na sintaxe e no comportamento da linguagem. O resultado foi a implementação do compilador Lua2IL, que lê arquivos de bytecodes Lua (gerados pelo interpretador a partir dos scripts) e gera *assemblies* com os bytecodes traduzidos para CIL. As assemblies geradas precisam de um pequeno runtime, uma assembly CLR com tipos e funções de suporte, mas são completamente independentes do interpretador Lua.

Lua2IL permite que os scripts compilados usem quase todos os recursos da linguagem Lua, com exceção da remoção dos pares das tabelas fracas, quando a chave (ou o valor) do par é coletada. Uma referência fraca do CLR não é notificada quando o objeto para o qual ela aponta é coletado, nem pode ser posta em uma fila pelo coletor de lixo (como na máquina virtual Java), logo não foi possível reproduzir este comportamento no código gerado por Lua2IL.

Os mesmos recursos que LuaInterface oferece para os scripts Lua são oferecidos para os scripts compilados por Lua2IL: o compilador Lua2IL é

PUC-Rio - Certificação Digital Nº 0210489/CA

um consumidor completo da CLS, e também tem a capacidade limitada de criar novas classes, com métodos implementados por código Lua, além de gerar uma nova classe para cada função Lua compilada; estas classes permitem que aplicações do CLR executem código Lua.

Na implementação da segunda abordagem a dificuldade principal foi a otimização do código gerado pelo compilador Lua2IL. Para conseguir um desempenho melhor do que o do interpretador Lua o compilador tem que gerar o código CIL de cada opcode, ao invés de simplesmente gerar uma chamada a um método em C# que executa a operação desejada. Esta e outras otimizações aumentam a complexidade do código gerado, dificultanto a depuraçao do compilador. Outras dificuldades foram a implementação de recursos que não têm suporte direto do CLR, como *upvalues*, co-rotinas e *weak tables*. Finalmente, como a segunda abordagem engloba a funcionalidade da primeira, também houve a dificuldade da geração dinâmica de código CIL para tratamento de eventos e criação de delegates e classes.

O desempenho do código gerado por Lua2IL foi comparado com o código gerado por outros dois compiladores de linguagens de script para o CLR: um compilador comercial da linguagem JScript, desenvolvido pela Microsoft, e o protótipo de um compilador da linguagem Python. O desempenho também foi comparado com o do interpretador Lua. O código gerado por Lua2IL apresentou desempenho superior aos outros, exceto no código envolvendo tabelas, quando o desempenho foi ligeiramente inferior ao do código JScript, e consideravelmente inferior ao do interpretador Lua. Esta parte do Lua2IL ainda tem bastante espaço para otimização, entretanto.

Também foi feita uma avaliação do tempo para chamada de métodos a partir do código compilado por Lua2IL, para comparação com LuaInterface. O desempenho foi superior, como esperado, levando de pouco menos da metade a pouco menos de um terço do tempo. A comparação com o código gerado pelo compilador JScript mostrou um tempo ligeiramente inferior, com o código JScript apresentando tempos cerca de 10% menores.

O nível de integração obtido pelas duas abordagens foi o mesmo. Uma vantagem da primeira abordagem é a sua maior facilidade de implementação. Lua2IL é apenas cerca de 50% maior que LuaInterface, em quantidade de linhas de código, mas isto se deve em grande parte à simplicidade da linguagem Lua e de sua máquina virtual, além do fato de Lua2IL ser um protótipo que não inclui a quase totalidade da biblioteca padrão de Lua e não compila código Lua diretamente, apenas bytecodes Lua.

Facilidade de implementação é uma vantagem importante, quando

PUC-Rio - Certificação Digital Nº 0210489/CA

se consideram as dificuldades enfrentadas na criação de compiladores de linguagens de script para o CLR. Qualquer linguagem que possua uma interface com código nativo, dinamicamente tipada, e uma maneira de estender dinamicamente o comportamento de seus objetos, pode ter uma ponte entre o interpretador da linguagem e o CLR com os mesmos recursos de LuaInterface. A maioria das linguagens de script possui esses pré-requisitos. Outra vantagem da abordagem implementada por LuaInterface é o reuso de todas as bibliotecas já existentes para a linguagem Lua, muitas delas implementadas em C.

A primeira abordagem, entretanto, tem a desvantagem de precisar de dois ambientes de execução diferentes, o interpretador Lua e o CLR. Isto dificulta a depuração das aplicações, e exige cuidado do programador no gerenciamento de memória, para evitar ciclos entre os dois ambientes: uma tabela Lua contendo uma referência para um objeto CLR que por sua vez contém uma referência para a tabela. Tais ciclos impedem que os objetos envolvidos sejam coletados pelos respectivos coletores de lixo.

A segunda abordagem, por sua vez, tem como vantagens o melhor desempenho, tanto na interface da linguagem com o CLR quanto na própria execução dos scripts, além de evitar os problemas com depuração e gerenciamento de memória da primeira abordagem. Uma desvantagem é a impossibilidade de se implementar eficientemente todos os recursos da linguagem Lua, devido a limitações na versão atual do CLR. Os scripts compilados também não podem usar as bibliotecas da linguagem Lua que já existem, com as bibliotecas tendo que ser reimplementadas em alguma linguagem do CLR. Finalmente, a segunda abordagem é de implementação mais difícil.

Até agora, a ênfase dos compiladores de linguagens de script para o CLR tem sido primeiro na geração estática (durante a compilação) de classes; esta ênfase termina por sacrificar os recursos dinâmicos das linguagens, que em geral permitem a criação e extensão de classes em tempo de execução, e termina por tornar a construção do compilador muito mais difícil (como no caso dos compiladores de Perl e Python) ou por exigir alterações na linguagem (o compilador JScript). A abordagem adotada na construção de Lua2IL, por outro lado, tem sua ênfase na reprodução todos os recursos da linguagem e no aspecto consumidor da interface dela com o CLR. Esta abordagem mostra que que é possível compilar o código de uma linguagem dinâmica para o CLR, obtendo um desempenho melhor do que o mesmo código sendo executado pelo interpretador da linguagem. A linguagem também passa a ter disponíveis todos os tipos do CLR, e com

PUC-Rio - Certificação Digital Nº 0210489/CA

os recursos de geração dinâmica de código do CLR ela também pode criar novas classes em tempo de execução.

Para o futuro, primeiro é preciso continuar o trabalho no compilador Lua2IL, com as seguintes tarefas, em ordem de importância: melhorar o desempenho de suas tabelas, usando as mesmas otimizações para vetores que o interpretador Lua implementa; fornecer uma implementação das funções da biblioteca padrão da linguagem Lua; e usar o próprio código Lua como base para a compilação, ao invés dos bytecodes, eliminando totalmente a necessidade do interpretador. Outro possível trabalho é o de modificar o CLR para corrigir o suporte às tabelas fracas, e melhorar a eficiência das co-rotinas com uma implementação que não dependa de sincronização entre threads.

PUC-Rio - Certificação Digital Nº 0210489/CA

Bibliografia

[1] DE FIGUEIREDO, L. H.; IERUSALIMSCHY, R. ; CELES, W.. **Lua — An Extensible Embedded Language**. Dr. Dobb's Journal, 21(12):26–33, 1996.

[2] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **Lua — An Extensible Extension Language**. Software: Practice and Experience, 26(6):635–652, 1996.

[3] OUSTERHOUT, J.. **Scripting: Higher Level Programming for the 21st Century**. IEEE Computer, 31(3):23–30, 1998.

[4] CERQUEIRA, R.; CASSINO, C. ; IERUSALIMSCHY, R.. **Dynamic Component Gluing Across Different Componentware Systems**. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS (DOA'99), 1999.

[5] CASSINO, C.; IERUSALIMSCHY, R.. **LuaJava — Uma Ferramenta de Scripting para Java**. In: SIMPóSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAçãO (SBLP'99), 1999.

[6] CASSINO, C.; IERUSALIMSCHY, R. ; RODRIGUEZ, N.. **LuaJava — A Scripting Tool for Java**. Technical report, Computer Science Department, PUC-Rio, 1999. Available at `http://www.tecgraf.puc-rio.br/~cassino/luajava/index.html`.

[7] ACTIVESTATE. **Release Information for the ActiveState Perl for .NET compiler**, 2000. Available at `http://www.activestate.com/Corporate/Initiatives/NET/Perl_release.html`.

[8] HAMMOND, M.. **Python for .NET: Lessons Learned**, 2000. Available at `http://www.activestate.com/Corporate/Initiatives/NET/Python_for_.NET_whitepaper.pdf`.

[9] INC., S.. **S#.NET Tech-preview Software Release**, 2000. Available at `http://www.smallscript.com/Community/calendar_home.asp`.

PUC-Rio - Certificação Digital Nº 0210489/CA

[10] MASCARENHAS, F.. **LuaInterface: User's Guide**. Computer Science Department, PUC-Rio, 2000. Available at http://www.inf.puc-rio.br/~mascarenhas/luainterface/manual-en.pdf.

[11] CERQUEIRA, R.; NOGUEIRA, L. ; DE MOURA, A. L.. **The LuaOrb Manual**. TeCGraf Computer Science Department, PUC-Rio, 2000. Available at http://www.tecgraf.puc-rio.br/luaorb/.

[12] MEIJER, E.; GOUGH, J.. **Technical Overview of the Common Language Runtime**. Technical report, Microsoft Research, 2002. Available at http://research.microsoft.com/~emeijer/Papers/CLR.pdf.

[13] GOUGH, J.. **Compiling for the .NET Common Language Runtime**. Prentice Hall, 2002.

[14] MICROSOFT. **ECMA C# and Common Language Infrastructure Standards**, 2002. Available at http://msdn.microsoft.com/net/ecma/.

[15] STUTZ, D.. **The Microsoft Shared Source CLI Implementation**, 2002. Available at http://msdn.microsoft.com/library/en-us/Dndotnet/html/mssharsourcecli.asp.

[16] DUBOIS, J.. **PerlNET — The Camel Talks .NET**. In: THE PERL 6 CONFERENCE, 2002. Available at http://conferences.oreillynet.com/presentations/os2002/dubois_update.ppt.

[17] XIMIAN. **The Mono Project**, 2003. Available at http://www.go-mono.com/.

[18] BOCK, J.. **.NET Languages**, 2003. Available at http://www.jasonbock.net/dotnetlanguages.html.

[19] IERUSALIMSCHY, R.. **Programming in Lua**. Lua.org, 2003.

[20] IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H. ; CELES, W.. **Lua 5.0 Reference Manual**. Technical Report 14/03, PUC-Rio, 2003. Available at http://www.lua.org.

[21] SHANKAR, A.. **Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API**. MSDN Magazine, 18(9), 2003. Available at http://msdn.microsoft.com/msdnmag/issues/03/09/CoroutinesinNET/default.aspx.

PUC-Rio - Certificação Digital Nº 0210489/CA

[22] JENSEN, J.. **LuaPlus 5.0 Distribution**, 2003. Available at http://wwhiz.com/LuaPlus/index.html.

[23] MICROSOFT. **Managed Extensions for C++ Migration Guide: Platform Invocation Services**, 2003. Available at http://msdn.microsoft.com/library/en-us/vcmxspec/html/vcmg_PlatformInvocationServices.asp.

[24] **Lua: user projects**, 2004. Available at http://www.lua.org/uses.html.

[25] MICROSOFT. **JScript .NET Language Reference**, 2004. Available at http://msdn.microsoft.com/library/en-us/jscript7/html/jsoriprogrammingwithjscriptnet.asp.

[26] BAGLEY, D.. **The Great Computer Language Shootout**, 2004. Available at http://dada.perl.it/shootout/.

PUC-Rio - Certificação Digital Nº 0210489/CA

A
Instruções da Máquina Virtual de Lua 5.0

Neste apêndice estão listadas as 35 instruções da máquina virtual de Lua 5.0. Instruções similares estão agrupadas. Algumas instruções têm argumentos que podem fazer referência tanto a um registrador quanto a uma constante. Se o valor do argumento é menor do que o número máximo de registradores, ele faz referência a um registrador. Se o valor do argumento é maior do que o número máximo de registradores subtrai-se este número do valor; o número obtido é posição da constante, no vetor de constantes, para o qual ele faz referência.

PUC-Rio - Certificação Digital Nº 0210489/CA

OP_MOVE A B Copia o valor do registrador B para o registrador A.

OP_LOADK A B Copia para o registrador A a constante armazenada na posição B do vetor de constantes.

OP_LOADBOOL A B C Armazena o valor `true` no registrador A se B for diferente de 0, senão armazena o valor `false`. Salta a próxima instrução se C for diferente de 1.

OP_LOADNIL A B Armazena o valor `nil` em todos os registradores entre o registrador A e o B (inclusive).

OP_GETUPVAL A B Copia para o registrador A o upvalue armazenado na posição B do vetor de upvalues.

OP_GETGLOBAL A B Copia para o registrador A o valor da tabela de globais cuja chave é o valor do registrador/constante B.
Se o valor da tabela for `nil` e a tabela possuir o meta-método `__index` chama o meta-método; o valor de retorno do meta-método é copiado para o registrador A.

OP_GETTABLE A B C Copia para o registrador A o valor da tabela armazenada no registrador B cuja chave é o valor registrador/constante C.
Se o valor da tabela for `nil` e a tabela possuir o meta-método `__index`

chama o meta-método; o valor de retorno do meta-método é copiado para o registrador A.

OP_SETGLOBAL A B Armazena o valor do registrador A na tabela de globais, usando o valor do registrador/constante B como chave.
Se a chave não existia e a tabela possui o meta-método `__newindex` chama o meta-método.

OP_SETUPVAL A B Copia o valor do registrador A para o upvalue armazenado na posição B do vetor de upvalues;

OP_SETTABLE A B C Armazena o valor do registrador/constante C na tabela armazenada no registrador A, com o registrador/constante C como chave.
Se a chave não existia e a tabela possui o meta-método `__newindex` chama o meta-método.

OP_NEWTABLE A B C Cria uma nova tabela e a armazena no registrador A, usando os valores dos registradores B e C como parâmetros para o tamanho inicial da tabela.

OP_SELF A B C Usado na chamada a métodos, copia o valor do registrador B para o registrador A+1, depois faz o equivalente a OP_GETTABLE A B C.

OP_ADD/OP_SUB/OP_MUL/OP_DIV/OP_POW A B C
Operações aritméticas binárias. Soma/subtrai/multiplica/divide/eleva à potência os valores dos registradores/constantes B e C, armazenando o resultado no registrador A.
Se um dos operandos não for numérico e possuir o meta-método apropriado (`__add`, `__sub`,`__mul`,`__div` ou `__pow`) chama o meta-método e copia seu valor de retorno para o registrador A.

OP_UNM A B Armazena a negação do valor do registrador B no registrador A.
Se o valor do registrador B não for numérico e possuir o meta-método `__unm` chama o meta-método e copia o valor de retorno para o registrador A.

OP_NOT A B Armazena a negação booleana do valor do registrador B no registrador A (`true` se o valor for `nil` ou `false` e `false` em caso contrário).

PUC-Rio - Certificação Digital Nº 0210489/CA

OP_CONCAT A B C Concatena os valores de todos os registradores entre B e C, inclusive, e armazena o resultado no registrador A.
Dois a dois os operandos são verificados; caso um dos dois não seja um número ou string e possua o meta-método `__concat` chama o meta-método com os dois operandos e o valor de retorno é usado.

OP_JMP A Salta A instruções para frente, se A for positivo, ou para trás, se for negativo.

OP_EQ/OP_LT/OP_LE A B C Se A for 0, e o valor do registrador/constante B for igual (para OP_EQ), menor (para OP_LT) ou menor ou igual (para OP_LE) ao valor do registrador/constante C, salta a próxima instrução.
Se A for 1, e o valor do registrador/constante B for diferente (para OP_EQ), maior ou igual (para OP_LT) ou maior (para OP_LE) que o valor do registrador/constante C, salta para a próxima instrução.
Caso os operandos não sejam números ou strings, tenham ambos o mesmo tipo e o mesmo meta-método apropriado para a operação (`__eq`, `__lt` ou `__le`) o meta-método é chamado para fazer a operação de comparação.

OP_TEST A B C Se C for 0, e o valor do registrador B for `nil` ou `false`, faz o equivalente a um OP_MOVE A B. Se C for 1, e o valor do registrador B não for `nil` nem `false`, também faz um equivalente a um OP_MOVE A B. Nos outros casos salta a próxima instrução.

OP_CALL A B C Executa a função armazenada no registrador A. A lista de argumentos começa com o valor do registrador A+1 e vai até o registrador A+B-1. Se B for igual a 1 a função é chamada sem argumentos, se for 0 a lista de argumentos vai do registrador A+1 até o topo da pilha (para o caso em que os argumentos são os valores de retorno de outra função).
O registrador C é o número de valores de retorno desejado menos um. Se C for 0 todos os valores de retorno são aproveitados, e o último valor retornado passa a ser o topo da pilha (para o caso em que os valores de retorno são usados como argumentos para outra função).
Na execução desta instrução a máquina virtual cria um novo registro de ativação para a função e guarda nele o endereço para retorno. O primeiro argumento passa a ser o registrador 0 da nova função.
Se o valor no registrador A não for uma função e possuir o meta-método `__call` todos os argumentos são movidos um registrador para cima; o valor do registrador A é copiado para o registrador A+1; o meta-método é copiado para o registrador A e chamado.

PUC-Rio - Certificação Digital Nº 0210489/CA

OP_TAILCALL A B C Implementa uma chamada a uma função com o retorno de todos os valores retornados por ela (ex. `return func(a,b,c)`). Funciona como OP_CALL A B C, mas reaproveita o registro de ativação e os registradores (o espaço na pilha de execução) da função atual. Os argumentos são copiados para os registradores, a partir do 0. Como o endereço de retorno da nova função passa a ser o da função atual, esta instrução equivale a retornar da função. O retorno implica no abandono do escopo definido pela função, logo os upvalues em aberto desse escopo são fechados.

OP_RETURN A B Retorna da função atual, com o primeiro valor de retorno no registrador A e os outros B-2 valores de retorno nos registradores seguintes. Se B for 1 a função não retorna nenhum valor, e se B for 0 ela retorna todos os valores entre o registrador A e o topo da pilha.
Os valores são copiados para os registradores da função para a qual se está retornando, a partir do que armazena a função chamada. Os valores a mais são descartados e os valores a menos completados com `nil`. Como o escopo definido pela função é abandonado os upvalues em aberto daquele escopo são fechados.

OP_FORLOOP A B A linguagem Lua tem dois tipos de laços *for*. O primeiro é o laço *for* numérico, cuja sintaxe é

```
for idx=inic,lim,incr do
  <bloco>
end
```

onde `idx` é a variável de controle do laço (local a ele), `inic` é o valor inicial, `lim` o limite e `incr` o incremento do a cada iteração.
A instrução OP_FORLOOP é o teste do laço *for* numérico. O registrador A contém a variável de controle, o registrador A+1 o limite do laço e o registrador A+2 o valor de incremento a cada iteração. OP_FORLOOP incrementa o índice e salta B instruções se ele ainda não atingiu o limite.

OP_TFORLOOP A B O segundo tipo da laço *for* é o laço com função de iteração, de sintaxe

```
for idx1,...,idxn in iter,estado,inic do
  <bloco>
end
```

PUC-Rio - Certificação Digital Nº 0210489/CA

onde `idx`k são as variáveis de iteração, `iter` é a função de iteração, `estado` é o valor que é sempre passado como primeiro argumento para `iter` e `inic` o valor inicial para a primeira variável de iteração.
Em cada iteração Lua chama a função de iteração passando o estado e a primeira variável de iteração. Os valores de retorno são armazenados nas variáveis de iteração. Se a primeira variável for `nil` o laço termina.
A instrução OP_TFORLOOP é o teste do laço *for* com função de iteração. O registrador A contém a função de iteração, os registrador A+1 contém o estado e o registrador A+2 a primeira variável de iteração. C é o número de valores de retorno da função de iteração.
OP_TFORLOOP executa a função de iteração e copia seus valores de retorno para os registradores a partir do A+2. Se o registrador A+2 for `nil` salta uma instrução (a instrução seguinte à OP_TFORLOOP é um salto para executar a próxima iteração do laço);

OP_TFORPREP A B Para compatibilidade com as versões anteriores de Lua, Lua 5.0 tem uma terceira sintaxe para laços *for*, específica para iteração em tabelas:

```
for chave,valor in tab do
  <bloco>
end
```

onde `tab` é a tabela que se quer iterar, `chave` e `valor` são variáveis locais ao laço, e que a cada iteração receberão um dos pares de chave e valor presentes na tabela.
OP_TFORPREP é uma instrução que converte um laço *for* para iteração em tabelas em um laço *for* com função de iteração.
OP_TFORPREP copia a tabela armazenada no registrador A para o registrador A+1, então copia a função de iteração `next`, armazenada em uma variável global, para o registrador A e salta B instruções (para uma instrução OP_TFORLOOP).
A tabela passa a ser o estado e o valor inicial é `nil`. `chave` e `valor` passam a ser as variáveis de iteração. A função `next` recebe uma tabela e uma de suas chaves como argumentos e retorna a próxima chave e seu valor.

OP_SETLIST/OP_SETLISTO A B Usados na construção de listas. Os itens que serão inseridos na tabela armazenada no registrador A estão a partir de registrador A+1.

PUC-Rio - Certificação Digital Nº 0210489/CA

No caso de OP_SETLIST, B é o índice do último elemento menos um. O número de itens para inserir é o valor de B módulo uma constante do interpretador, o máximo de itens inseridos de uma só vez, mais um. O índice do primeiro item é o índice do último menos o número de itens para inserir.
OP_SETLISTO é usado quando o último item é uma chamada a uma função. Neste caso, B é o índice do primeiro valor de retorno da função. O número de itens a inserir é o número de elementos entre o registrador A e o topo da pilha. O índice do primeiro item é o valor de B menos B módulo o máximo de itens inserido de uma só vez, mais um.

OP_CLOSE A Fecha os upvalues do escopo atual que estão em aberto (todos os upvalues na apontando para valores na pilha acima da posição do registrador A).

OP_CLOSURE A B Instancia a B-ésima (começando em 0) função declarada pela função atual e a armazena no registrador A. Se a função possui upvalues a instrução é seguida por instruções para inicializar o vetor de upvalues, uma instrução para cada upvalue da função.
Estas instruções podem ser OP_MOVE, se o upvalue aponta para uma variável local do escopo atual (no registrador B da instrução OP_MOVE); nesse caso primeiro se procura o upvalue entre os que já estão em aberto, e caso não exista é criado e posto na lista de upvalues em aberto.
As instruções também podem ser OP_GETUPVAL, se o upvalue aponta para uma variável local de um escopo externo à função atual (um dos upvalues da função atual). Neste caso o upvalue na posição B do vetor de upvalues da função atual é copiado para o vetor de upvalues da função instanciada por OP_CLOSURE.
O processamento das instruções para inicialização do vetor de upvalues é parte da execução da instrução OP_CLOSURE, portanto o interpretador salta estas instruções.

PUC-Rio - Certificação Digital Nº 0210489/CA