3
APIs de linguagens de script

Interfaces providas por linguagens de script são usualmente compreendidas como "APIs de extensão": elas servem para estender a máquina virtual com recursos não oferecidos por esta, ou ainda para estender uma aplicação externa com os recursos oferecidos pelo ambiente de execução da linguagem, embutindo-o nesta aplicação. O primeiro cenário é o utilizado no modelo de programação onde a coordenação de alto nível é realizada em uma linguagem interpretada e módulos compilados em linguagens como C e C++ são usados para acesso a bibliotecas externas ou implementação de partes onde o desempenho é crítico. O segundo cenário, via de regra, irá englobar também o primeiro, ao expor à máquina virtual embutida extensões que a permitam comunicar-se com a aplicação hospedeira.

Ambos os cenários envolvem os mesmos problemas gerais: transferência de dados entre uma linguagem e outra, incluindo como permitir à linguagem de script manipular estruturas declaradas em C e vice-versa; tratar a diferença entre os modelos de gerência de memória, mais especificamente a interação entre a coleta de lixo na máquina virtual e a desalocação explícita em C; a chamada de funções declaradas pela linguagem de script a partir de C; e o registro de funções C para que possam ser invocadas a partir de scripts. As seções seguintes discutem as principais questões envolvidas na comunicação entre código C e de linguagens de script, e apresentam as abordagens empregadas pelas APIs de Python, Ruby, Java, Lua e Perl no tratamento destas questões. Cada seção conclui com uma comparação onde as diferentes características apresentadas na exposição de cada linguagem são revistas lado a lado e assim postas em perspectiva.

3.1
Transferência de dados

O principal complicador na interação entre linguagens de programação não é a diferença de sintaxe ou semântica das construções de fluxo de controle, mas a representação dos dados. Na comunicação entre código escrito em duas linguagens diferentes, dados trafegam de várias formas: como parâmetros,

atributos de objetos, elementos de estruturas de dados, etc.

Freqüentemente, o formato como estes dados são representados difere. Nestes casos, há três alternativas para realizar a transferência de dados entre as duas linguagens. A mais simples é expor o dado à linguagem de destino como uma entidade opaca. A linguagem de destino recebe apenas algum tipo de *handle* que permite identificar unicamente o dado em operações requisitadas posteriormente. Esta abordagem é útil, por exemplo, se uma linguagem está apenas armazenando os dados para a outra, a fim de aproveitar estruturas de dados de mais alto nível que a linguagem ofereça.

Outra abordagem envolve realizar algum tipo de conversão dos dados do sistema de tipos de uma linguagem para o da outra. A duplicação que ocorre na conversão limita a aplicabilidade deste método, restringindo o seu uso tipicamente para tipos numéricos e, em menor grau, strings. Finalmente, a linguagem de origem pode explicitamente oferecer facilidades na linguagem de destino para a manipulação destes dados, ou seja, uma linguagem oferecer uma API para a outra. A diferença entre esta abordagem e a primeira é que, enquanto naquela o conteúdo do dado se mantém opaco, nesta a própria API define uma forma de manipular o seu conteúdo.

C, por ser orientada à manipulação de ponteiros e estruturas, possui um conjunto pequeno de tipos básicos. Além disso, C é bastante liberal em relação à representação interna dos tipos estruturados, cabendo às diferentes plataformas definirem interfaces binárias (*application binary interfaces*, ABIs). Assim, mesmo em casos onde é possível ligar código C diretamente utilizando tipos básicos compatíveis e convenções de chamada apropriadas (como em Free Pascal ou em diversos compiladores Fortran), uma biblioteca de *bindings* é usualmente necessária para tornar mais conveniente a manipulação de tipos mais complexos.

Até nos tipos numéricos fundamentais, há vários cuidados que devem ser tomados. Algumas linguagens, como Smalltalk e Ruby, realizam conversão automática de inteiros para "inteiros grandes" (*bignums*). Em Ruby, particularmente, os inteiros primitivos têm 1 bit a menos de precisão do que o tamanho de palavra da máquina. Pode haver ainda a necessidade de conversão de *endianness* e formato de números de ponto flutuante.

Para tipos como strings, o tamanho dos valores traz ainda preocupações com desempenho. Em diversos casos a representação interna usada para strings é a mesma que a usada em C, então uma opção é simplesmente passar ao código C um ponteiro para o endereço onde a string está armazenada, o que evita a cópia de dados, sob risco de permitir ao programa C modificar o conteúdo da string. Expor ao código C ponteiros para endereços de memória dentro

do ambiente de execução da outra linguagem pode trazer ainda problemas de concorrência, caso o ambiente em questão utilize múltiplas *threads*.

Ao expor ao código C dados de tipos estruturados, a conversão para um tipo de dado nativo de C, em muitos casos, não é uma opção. Além da questão da quantidade de dados, tipos estruturados em C são definidos estaticamente, não servindo assim para representar convenientemente dados de estruturas dinâmicas, como objetos que podem ganhar ou perder atributos e até mesmo mudar de classe em tempo de execução. Mesmo em linguagens com tipos estáticos, como Java, a cópia de objetos não costuma ser uma opção interessante em função do volume de dados. A cópia de objetos estruturados costuma se restringir a operações específicas como manipulação de arrays de elementos primitivos.

A alternativa para permitir que código C opere sobre dados estruturados, então, é fornecer a ele uma API que exponha, em forma de funções, as operações definidas sobre os tipos em questão. Isto evita também a necessidade de controlar a consistência entre duas cópias de uma mesma estrutura. Problemas de consistência, entretanto, podem ocorrer caso a API permita ao código C armazenar ponteiros para objetos da linguagem – isto torna necessário ao programador gerenciar explicitamente a sincronia entre os ponteiros e o ciclo de vida dos objetos, que podem estar sujeitos a coleta de lixo. A Seção 3.2 discute esta questão em mais detalhe.

3.1.1 Python

Todos os valores na máquina virtual de Python são representados como objetos, mapeados para a API de C como a estrutura `PyObject` (van Rossum 2006a). Tipos mais específicos como `PyStringObject`, `PyBooleanObject` e `PyListObject` são `PyObject`s por equivalência estrutural, isto é, podem ser convertidos através de um *cast* C. Refletindo o modelo de tipagem dinâmica de Python, as funções da API usam `PyObject*` como tipo sempre que se referem a objetos Python, mesmo quando são projetadas para atuar sobre valores Python de tipos mais específicos, como por exemplo a função `PyString_Size`, que retorna o tamanho de uma string. Cada tipo específico possui uma função de verificação na API, como `PyNumber_Check` e `PyDict_Check`.

Python é uma linguagem fortemente tipada: cada objeto é amarrado a um tipo. Tipos são representados por estruturas `PyTypeObject`, que também são estruturalmente equivalentes a `PyObject`. Cada tipo Python possui um `PyTypeObject` pré-definido na API, como `PyString_Type`, `PyBoolean_Type` e

`PyList_Type`. `PyObject_TypeCheck` compara o tipo de um `PyObject` a um `PyTypeObject` passado como parâmetro.

Para a conversão de dados de C para Python, a linguagem oferece uma série de funções que recebem valores de tipos primitivos de C como parâmetro, como `PyString_FromStringAndSize` e `PyFloat_FromDouble`. Cada uma destas funções retorna ao código C um ponteiro para um novo objeto `PyObject`. Strings passadas são copiadas por Python. O exemplo a seguir ilustra a criação um objeto Python através da conversão de um valor C:

```
PyObject* s = PyString_FromString("hello");
```

Os dois exemplos abaixo, equivalentes, ilustram a verificação de tipo através da API, primeiro através de uma função de conveniência, e depois explicitamente, comparando o tipo da string Python com `PyString_Type`:

```
if (PyString_Check(s)) printf("Sim.\n");
if (PyObject_TypeCheck(s, PyString_Type)) printf("Sim.\n");
```

Para o retorno de dados de Python para C, um conjunto complementar de funções é oferecido, mapeando os tipos básicos de Python de volta a tipos de C. Alguns exemplos destas funções que recebem um ponteiro para um `PyObject` como parâmetro e retornam o dado correspondente em C são `PyLong_AsUnsignedLong` e `PyString_AsStringAndSize`. Diferentemente das funções de entrada, nas funções de saída não há cópia de strings: as strings retornadas são ponteiros para a área armazenada internamente por Python. A documentação recomenda não modificar o conteúdo da string exceto no caso de esta área de memória haver sido retornada por uma chamada a `PyString_FromStringAndSize(NULL,` *tamanho*`)` (van Rossum 2006c). Desta forma, é possível alocar uma string para armazenamento em Python e preenchê-la posteriormente através de código C, como no exemplo a seguir:

```
/* alocando uma string não-inicializada em Python */
PyObject* obj = PyString_FromStringAndSize(NULL, 51);
/* obtendo o ponteiro para a área de memória da string */
char* s = PyString_AsString(obj);
/* Agora, podemos preencher a string em C. Um exemplo qualquer: */
for (int i = 0; i < 5; i++, s+=10)
   snprintf(s, 11, "[%-8d]", random());
```

Para alguns de seus tipos básicos que não possuem correspondente direto em ANSI C 89, Python define tipos em C equivalentes: `Py_UNICODE` e `Py_complex`. Estes tipos foram adicionados com o objetivo de expor a representação interna dos dados usada por Python aos módulos para manipulação numérica ou de texto Unicode implementados em C, evitando conversões freqüentes de e para `PyObject`.

Python oferece ainda algumas versões de funções de conversão para C implementadas como macros sem a verificação de tipo, assumindo que o `PyObject` passado será compatível, oferecendo melhor desempenho às custas de segurança. Estas funções podem ser identificadas pelos nomes em maiúsculas. Entre as macros de conversão oferecidas estão `PyString_AS_STRING`, `PyInt_AS_LONG` e `PyUnicode_AS_UNICODE`.

Além de funções para conversão de tipos entre Python e C, a API de Python oferece ainda algumas funções de conversão entre tipos de Python. Estas funções recebem um `PyObject` como parâmetro e retornam um novo `PyObject` com o resultado da conversão, e são equivalentes a funções Python que realizam estas conversões (na verdade chamadas a tipos `PyTypeObject` que respondem ao método `__call__`). Por exemplo, a função `PyObject_Str` é equivalente à função Python `str`.

Em Python, objetos são armazenados em módulos, que são espaços de nomes declarados globalmente, ou como atributos de objetos. Variáveis são armazenadas em um *ambiente*, representado como um dicionário. Funções como `PyRun_File` recebem, entre seus parâmetros, um dicionário de variáveis globais e outro de variáveis locais. O conjunto de variáveis e funções globais é representado como o dicionário do módulo `__main__`. Objetos *built-in* são acessíveis através do módulo `__builtin__`. Por exemplo, para obter o objeto `str`, obtemos inicialmente uma referência ao módulo `__builtin__` com a função `PyImport_AddModule` e em seguida o dicionário do módulo com a função `PyModule_GetDict`.

```
PyObject* builtins_module = PyImport_AddModule("__builtin__");
PyObject* builtins = PyModule_GetDict(builtins_module);
PyObject* str = PyDict_GetItemString(builtins, "str");
```

Em Python, `str` é um objeto chamável, que atua como a função de conversão para strings. Assim, uma vez que obtivemos uma referência para o `PyObject` correspondente a `str`, a chamada seguinte equivale a chamar `PyObject_Str` sobre um objeto Python qualquer `obj`:

```
/* Esta é uma função vararg que recebe como argumentos adicionais
   uma lista terminada em NULL de PyObjects a serem passados à
   função Python indicada no primeiro argumento. */
PyObject* result = PyObject_CallFunctionObjArgs(str, obj, NULL);
```

O armazenamento de dados de C no espaço de objetos de Python pode ser feito de duas formas. Uma maneira é criar um objeto do tipo `CObject` encapsulando um ponteiro C qualquer, construindo assim um valor opaco para

Python. As funções de construção deste objetos deste tipo permitem associar ao dado uma função C a ser chamada quando o `CObject` for desalocado. Segundo a documentação de Python, `CObject`s têm como objetivo principal permitir a passagem de dados em C de um módulo de extensão para outro (van Rossum 2006c).

A outra forma é declarar novos tipos Python através de estruturas em C. Em C, um tipo Python é descrito em duas partes: um tipo *struct,* a partir do qual instâncias do tipo serão produzidas, e uma instância da struct `PyTypeObject`, que descreverá o tipo para Python. O exemplo a seguir ilustra a criação de um novo tipo Python em C. Inicialmente, temos `point`, que será o tipo C das instâncias dos objetos:

```
typedef struct {
   PyObject_HEAD
   int x, y;
} point;
```

A macro `PyObject_HEAD` garante equivalência estrutural com `PyObject`. Quando funções retornarem o objeto ao código C como um `PyObject*`, este pode ser convertido de volta para `point` via cast, dando assim acesso aos atributos `x` e `y`. Definiremos também uma função que opera sobre objetos deste tipo:

```
PyObject* point_distance(point* p) {
    return PyFloat_FromDouble( sqrt(p->x*p->x + p->y*p->y) );
}
```

A função é definida com tipo de retorno `PyObject`* para que possa ser registrada na máquina virtual de Python. Para associar a função ao tipo Python, iremos inicialmente armazená-la em um array de estruturas `PyMethodDef`, que listará os métodos do tipo:

```
static PyMethodDef point_methods[] = {
   { "distance", (PyCFunction) point_distance, METH_NOARGS },
   { NULL }
};
```

Para que os atributos do tipo sejam visíveis a partir de Python, precisamos implementar uma rotina de acesso, que recebe o objeto e o nome do atributo acessado. A sua implementação é dada a seguir:

```
PyObject* point_getattr(PyObject* self, char* name) {
   if (strcmp(name, "x") == 0)
      return PyInt_FromLong(((point*)self)->x);
```

```
    else if (strcmp(name, "y") == 0)
        return PyInt_FromLong(((point*)self)->y);
    else
        return Py_FindMethod(point_methods, self, name);
}
```

Uma vez registrada na descrição do tipo, esta função será responsável por retornar os seus atributos e métodos. Assim, podemos expor ao ambiente Python atributos armazenados na struct C. A função `Py_FindMethod` localiza uma função no array passado como parâmetro e a retorna como um método[1].

Finalmente, definiremos `point_type`, que será o `PyTypeObject` que descreve o tipo Python relativo a eles[2]:

```
static PyTypeObject point_type = {
    PyObject_HEAD_INIT(NULL)
    /* O nome da classe */
    .tp_name = "point",
    /* O tamanho da área de memória a ser alocada */
    .tp_basicsize = sizeof(point),
    /* A função de acesso a atributos */
    .tp_getattr = point_getattr,
    /* Classe não requer tratamentos especiais */
    .tp_flags = Py_TPFLAGS_DEFAULT
};
```

Novamente, uma macro foi usada no início da definição do tipo para garantir equivalência estrutural. `PyTypeObject` possui vários outros campos, mas os manteremos como `NULL` de modo que sejam preenchidos com valores default quando da construção do tipo em tempo de execução. O tipo `PyTypeObject` contém uma série de campos que permitem descrever o comportamento do tipo declarado. No campo `tp_getattr` de `point_type`, especificamos que a função C a ser usada para tratar acesso a atributos será `point_getattr`. Especificamos `Py_TPFLAGS_DEFAULT` no campo de flags para indicar uma classe de comportamento padrão, sem a necessidade de tratamentos especiais como verificação de ciclos na coleta de lixo.

Embora a representação em memória de objetos Python do tipo definido pelo usuário sejam instâncias de `point`, para criar um novo objeto não basta alocar uma instância da struct `point` e usá-la como `PyObject` via cast. É preciso inicializar o objeto para que este seja registrado no mecanismo de coleta de lixo e tenha os campos de seu cabeçalho `PyObject` devidamente inicializados. A

[1] O registro de funções Python será discutido em detalhe na Seção 3.4.1.

[2] Para maior brevidade, apresentamos o exemplo usando a sintaxe para descrição de structs de C99, evitando-nos de listar os elementos a serem inicializados com `NULL`, já que a struct `PyTypeObject` possui 54 campos ao todo.

alocação em C de novos objetos de um tipo definido pelo usuário deve ser feita através da macro `PyObject_New`, que recebe como parâmetros o tipo da struct a ser alocada e o `PyTypeObject` correspondente ao tipo. Antes de alocar objetos do tipo em C, deve-se completar a inicialização de `point_type` em tempo de execução. A documentação recomenda inicializar em tempo de execução a função padrão de construção de objetos, `PyType_GenericNew`, por razões de portabilidade (van Rossum 2006a). Finalmente, o preenchimento dos campos não definidos na declaração da struct é feita pela função `PyType_Ready`.

```
point_type.tp_new = PyType_GenericNew;
if (PyType_Ready(&point_type) < 0) return;
```

A partir daí, instâncias podem ser criadas com `PyObject_New`, como no exemplo abaixo:

```
/* Cria uma instância */
point* a_point = PyObject_New(point, &point_type);
a_point->x = 100; a_point->y = 200;
/* Armazena a instância na global Python P,
   assume que o dicionário de globais já foi armazenado em globals */
PyDict_SetItemString(globals, "P", (PyObject*) a_point);
```

Uma vez declarado em C, este valor pode ser usado em Python:

```
print 'P.x = ' + str(P.x)
print 'P.y = ' + str(P.y)
print 'd   = ' + str(P.distance())
```

A API de Python possui um grande número de funções para manipulação de tipos pré-definidos da linguagem. *Tuplas* merecem menção especial no que tange à transferência de dados entre Python e C, pois são usadas em diversos contextos: na passagem de parâmetros em chamadas a funções Python a partir de C, no recebimento dos parâmetros de entrada em funções C e também na passagem e recebimento de múltiplos valores de retorno, como será visto nas Seções 3.3.1 e 3.4.1.

Como tuplas são usadas com freqüência como "ponte" entre Python e C, a API possui uma função de conveniência, `PyArg_ParseTuple`, que evita que o acesso aos elementos da tupla e a verificação dos seus tipos seja feita item a item pelo programador. Trata-se de uma função C *vararg* que recebe como parâmetros a tupla, uma string indicando os tipos dos parâmetros esperados e os endereços onde os valores, convertidos para tipos C, devem ser armazenados. A função define uma sintaxe para os indicadores de tipo especificados na string passada e os tipos C correspondentes. Por exemplo:

`"s#"` indica que a tupla deve conter um objeto Python de tipo `string` ou `Unicode` e que dois parâmetros devem ser passados à função C, de tipos `const char**` e `int*`, que irão retornar o ponteiro para a string e o seu tamanho. Em um exemplo mais elaborado, `"iiO!|(dd)"` indica que a função espera dois endereços de inteiros (`"ii"`), seguido do endereço de um ponteiro `PyObject` (`"O"`) e de um objeto `PyTypeObject` a ser usado na verificação do tipo do objeto recebido (`"!"`) e opcionalmente (`"|"`), dois endereços de valores `double` passados via Python através de uma outra tupla (`"(dd)"`).

De forma similar, a API de Python possui a função `Py_BuildValue`, que permite a construção de objetos estruturados, como tuplas, listas e dicionários, em uma só chamada. Esta função é freqüentemente usada tanto para a construção da tupla de parâmetros ao chamar funções como para valores de retorno. A sintaxe da string de parâmetros é similar à de `PyArg_ParseTuple`, mas possui um conjunto diferente de indicadores de tipo, além de permitir descrever listas e dicionários. Por exemplo, a seguinte chamada cria uma lista contendo um inteiro, um número de ponto flutuante e um dicionário contendo um elemento de chave string e valor inteiro:

```
PyObject* lista = Py_BuildValue("[id{si}]", 123, 12.30, "foo", 1234);
```

Isto equivale à seguinte construção em Python:

```
lista = [123, 12.30, {"foo": 1234}]
```

3.1.2 Ruby

Para a comunicação de dados entre Ruby e C, a API de Ruby define um tipo de dados em C chamado `VALUE`, que representa um objeto Ruby. `VALUE` pode representar tanto uma referência para um objeto (isto é, um ponteiro para a *heap* de Ruby) como um valor imediato. Em particular, as constantes `Qtrue`, `Qfalse` e `Qnil` são definidas como valores imediatos, permitindo a comparação destas em C usando o operador `==`.

Para a verificação de tipos, Ruby disponibiliza as macros `Check_Type` e `TYPE`. `Check_Type` permite comparar o tipo de valores a constantes que descrevem os tipos básicos de Ruby como `T_OBJECT` e `T_STRING`. `TYPE` retorna a constante relativa ao tipo de um valor passado. Para a verificação da classe de um objeto, devemos usar `rb_class_of`.

Para a transferência de valores numéricos, a conversão entre C e Ruby é feita através de macros como `INT2NUM` e de funções como `rb_float_new`, que recebem ou retornam `VALUE`s.

Para a passagem de strings para Ruby a partir de C, são oferecidas as funções `rb_str_new`, que recebe um ponteiro e um argumento numérico de tamanho, de modo a permitir a passagem de strings contendo caracteres nulos, e `rb_str_new2`, que assume uma string padrão de C, com o caracter nulo como terminador. Estas funções fazem uma cópia da string C para o espaço de Ruby. `VALUE`s que apontam para strings de Ruby permitem acessar e alterar o seu conteúdo através do *cast* `RSTRING(uma_string)->ptr`. Todavia, a API recomenda o uso da macro `StringValue`, que retorna o próprio `VALUE` passado caso este seja uma string, ou um novo `VALUE` da classe `String` produzido através do método de conversão `to_s` aplicado ao objeto passado (ou ainda causa uma exceção `TypeError` caso a conversão não seja possível).

```
void mostra_valor(VALUE obj) {
   const char* s;
   if (TYPE(obj) == T_STRING) {
      /* Faria um acesso ilegal se TYPE(obj) != T_STRING */
      s = RSTRING(obj)->ptr;
   } else {
      /* Funciona para qualquer tipo que aceite obj.to_s,
         dispara uma exceção em caso contrário */
      s = StringValue(obj);
   }
   printf("Valor: %s\n", s);
}
```

Sob a justificativa de aumentar o desempenho no acesso, alguns outros tipos de Ruby como `Array`, `Hash` e `File` permitem acesso de baixo nível aos membros das estruturas utilizadas na implementação dos objetos. Por exemplo, `RARRAY(um_array)->len` permite ler o tamanho de um array diretamente. A recomendação da API é utilizar este tipo de acesso somente para leitura, já que a alteração destes valores pode facilmente tornar o estado interno dos objetos inconsistente.

Para o armazenamento de dados de C no espaço de objetos de Ruby, a API oferece uma macro, `Data_Wrap_Struct`, que recebe um ponteiro C e cria um objeto Ruby que encapsula este ponteiro. O ponteiro pode ser acessado a partir de código C usando `Data_Get_Struct`, mas não a partir de Ruby. Em `Data_Wrap_Struct` é passada também uma função C a ser executada quando o objeto for coletado. Por exemplo, criemos uma classe `Point`, similar ao tipo definido em Python na seção anterior. Inicialmente definiremos um tipo C:

```
typedef struct {
   int x, y;
} point;
```

Funções de alocação (`point_alloc`) e desalocação (`point_free`) para a classe `Point` são dadas a seguir:

```
void point_free(void* p) {
   free(p);
}

VALUE point_alloc(VALUE point_class) {
   point* p = malloc(sizeof(point));
   /* O segundo argumento é a função de marcação para coleta de lixo
      (NULL aqui pois o tipo não armazena VALUEs), cf. Seção 3.2.2 */
   return Data_Wrap_Struct(point_class, NULL, point_free, p);
}
```

Note que `Data_Wrap_Struct` faz uso de um `VALUE` que representa a classe `Point` em Ruby. Classes são criadas em C com a função `rb_define_class`. Esta função recebe uma string C com o nome da nova classe e um `VALUE` a ser usado como superclasse (como por exemplo a constante `rb_cObject`, que representa a classe Ruby `Object`) e retorna um `VALUE` representando a nova classe. Para classes como `Point`, cujas instâncias irão conter dados de C, é possível registrar uma função C responsável por realizar a alocação de memória das instâncias usando a função `rb_define_alloc_func`. A criação da classe e o registro da função de alocação, então, se dão da seguinte forma:

```
VALUE point_class = rb_define_class("Point", rb_cObject);
rb_define_alloc_func(point_class, point_alloc);
```

Como em código Ruby, a declaração de atributos de objetos é feita no método `initialize`, que pode ser implementado em C:

```
VALUE point_initialize(VALUE self, VALUE x, VALUE y) {
   point* p;
   Data_Get_Struct(self, point, p);
   p->x = NUM2INT(x);
   p->y = NUM2INT(y);
   return self;
}
```

O método é registrado na classe em tempo de execução com a função `rb_define_method` (o registro de funções C em Ruby será discutido em detalhe na Seção 3.4.2).

```
rb_define_method(point_class, "initialize", point_initialize, 2);
```

Para que a cópia de objetos através dos métodos Ruby `dup` e `clone` trate corretamente os dados armazenados via C, é preciso ainda registrar o método `initialize_copy`. Uma possível implementação em C é dada abaixo:

```
VALUE point_initialize_copy(VALUE copy, VALUE orig) {
   point* p_copy;
   point* p_orig;
   /* Ruby pode chamar esta função com o mesmo objeto nos dois
      parâmetros; nesse caso, ignore a chamada e retorne o objeto */
   if (copy == orig) return copy;
   /* Obter os ponteiros armazenados nos objetos */
   Data_Get_Struct(orig, point, p_orig);
   Data_Get_Struct(copy, point, p_copy);
   /* Cópia da ''parte C'' do objeto */
   p_copy->x = p_orig->x;
   p_copy->y = p_orig->y;
   /* Retorna a cópia */
   return copy;
}
```

Completemos o exemplo com uma função C implementando o método `distance` como feito na seção anterior para Python:

```
VALUE point_distance(VALUE self) {
   point* p;
   Data_Get_Struct(self, point, p);
   return rb_float_new( sqrt(p->x*p->x + p->y*p->y)) );
}
```

Estas funções também são registradas como métodos de `Point`:

```
rb_define_method(point_class, "initialize_copy",
                 point_initialize_copy, 1);
rb_define_method(point_class, "distance", point_distance, 0);
```

A função `rb_class_new_instance` produz novos objetos Ruby que são instâncias da classe, recebendo um array C de `VALUE`s a ser passados na inicialização e o `VALUE` da classe.

O acesso a variáveis Ruby se dá através da família de funções `rb_*_get`, que retornam os `VALUE`s relativos a atributos de objetos ou classes, variáveis globais e constantes. Para cada uma destas há uma função `rb_*_set` análoga[3]. As funções `rb_iv_get` e `rb_ivar_get`, por exemplo, obtêm atributos de objetos (*instance variables*). A primeira forma usa strings C como nomes, a segunda usa `ID`s, identificadores que substituem strings internalizadas na tabela de símbolos de Ruby, que podem ser obtidos usando a função `rb_intern`. De fato, `ID`s correspondem ao tipo *símbolo* de Ruby, que na prática são strings imutáveis. O seguinte exemplo obtém o valor de uma variável global `g` e a

[3]Constantes podem ser criadas com o valor `Qundef` e terem o seu valor definido posteriormente com `rb_const_set`, porém uma única vez.

atribui ao campo `c` de um objeto, e depois altera o valor da variável global para zero:

```
/* Obtém variável global */
VALUE g = rb_gv_get("g");
/* Atribui ao campo c do objeto obj */
VALUE obj = rb_gv_get("obj");
/* Equivale a: rb_ivar_set(obj, rb_intern("c"), g); */
rb_iv_set(obj, "c", g);
/* Zera a variável global */
rb_gv_set("g", INT2NUM(0));
```

IDs nunca são coletados: observamos que a tabela de símbolos não é zerada mesmo com `ruby_finalize`. Assim, uma aplicação C que oferece uma interface para scripting criando ambientes supostamente isolados, com `ruby_init` e `ruby_finalize` cercando cada execução de script, pode ter o seu consumo de memória aumentado indefinidamente à medida que os scripts criam símbolos.

3.1.3 Java

A JNI define no cabeçalho `jni.h` tipos em C equivalentes a cada um dos tipos primitivos de Java (`jint` para `int`, `jfloat` para `float`, e assim por diante). Os "tipos de referências", como classes e objetos, são expostos a C como referências opacas, instâncias de `jobject`. Strings e arrays também são objetos em Java e são portanto expostos como instâncias de `jobject`. Entretanto, a JNI define como conveniência alguns tipos em C que agem como "subtipos" de `jobject`: `jclass`, `jstring`, `jthrowable`, `jarray`, `jobjectArray`, além de um tipo array para cada tipo primitivo (`jbooleanArray`, `jbyteArray`, etc.). O tipo `jvalue` é uma *union* dos tipos primitivos e de referências. O valor C `NULL` equivale a `null` em Java.

Diferentes métodos são utilizados para a leitura de tipos primitivos, strings, arrays e outros objetos. A leitura do conteúdo de uma `jstring` em C requer a conversão do formato interno usado por Java, UTF-16. A API oferece uma função utilitária que aloca uma string contendo a representação do texto em UTF-8 (formato compatível com ASCII), `GetStringUTFChars`. Esta string deve ser posteriormente desalocada com `ReleaseStringUTFChars`. A função `GetStringChars` permite acesso direto à string em formato UTF-16; ela possui um parâmetro de saída que indica se a string retornada é o *buffer* interno da JVM ou uma cópia. Ao mesmo tempo que isto permite ao código C evitar duplicação da string nos casos onde deseja-se modificá-la e

a JVM tenha optado por retornar uma cópia, tal parâmetro expõe na API questões de baixo nível da gerência de strings na JVM. Alternativamente, as funções `GetStringRegion` e `GetStringUTFRegion` realizam a cópia da string para um buffer pré-alocado pelo programador. `GetStringCritical` permite obter o ponteiro para o buffer interno da JVM, mas isto envolve cuidados especiais em relação à coleta de lixo, discutidos na Seção 3.2.3.

Arrays de elementos primitivos são tratados de forma similar a strings, diferentemente de arrays de objetos[4]. Há funções para realizar a cópia de arrays (`Get/Set`*`[tipo]`*`ArrayRegion`), funções que retornam ponteiros para o array podendo ou não realizar cópias, de forma análoga a `GetStringChars` (`Get/Release`*`[tipo]`*`ArrayElements`) e que podem acessar o buffer interno da JVM diretamente, como em `GetStringCritical` (`Get/ReleasePrimitiveArrayCritical`). Para arrays de objetos, não é possível obter um ponteiro para o buffer interno do array, mas somente acessar e modificar os seus elementos um a um, sob forma de referências `jobject`, com `Get/SetObjectArrayElement`.

A obtenção de valores de atributos se dá através de funções como `GetObjectField` e `GetStaticField`, que retornam referências do tipo `jobject`. Para cada um dos tipos primitivos existe uma chamada equivalente, como `GetIntField` e `GetStaticIntField`. Assim como Ruby, a API de Java define um tipo C específico para evitar o uso freqüente de strings C na descrição de campos. Todavia, enquanto Ruby utiliza `ID`s que são simplesmente strings internalizadas, em Java os identificadores de campo, do tipo `jfieldID`, contêm informação de tipo e são específicos para o campo de uma determinada classe. Estes valores são obtidos com uma chamada a `GetFieldID`, que recebe entre seus parâmetros uma string chamada de "descritor de campo JNI" com uma sintaxe especial. Por exemplo, o tipo Java `int[][]` é descrito com `"[[I"` e o tipo `java.lang.String` como `"Ljava/lang/String;"`[5]. É possível ainda obter um `jfieldID` a partir de um objeto `java.lang.reflect.Field` usando a função `FromReflectedField`.

As chamadas à JNI têm o formato `(*J)->`*`função`*`(J, ...)`: funções da JNI são acessadas através de ponteiros para função armazenados em uma tabela apontada pela estrutura `JNIEnv`, que por sua vez é propagada nas chamadas. O objetivo destes dois níveis de indireção é desacoplar a ligação das chamadas no código C e a biblioteca que implementa a JNI, permitindo ligar o código

[4]Arrays multi-dimensionais são considerados "arrays de arrays" e, portanto, são também arrays de objetos.

[5]Este é outro ponto onde detalhes da implementação transparecem na API. Não por coincidência, esta sintaxe é a mesma usada na representação interna de tipos em *bytecodes* da JVM.

```
public class ExemploJNI {
   private String[] elementos = { "Terra", "Ar", "Fogo", "Agua" };

   /* Declaração do método implementado externamente */
   private native void segundoElemento();

   public static void main(String[] args) {
      /* Cria uma instância e invoca o método nativo */
      new ExemploJNI().segundoElemento();
   }

   static {
      /* Carrega o código externo na JVM que
         implementará o método segundoElemento */
      System.loadLibrary("ExemploJNI");
   }
}
```

Figura 3.1: Classe Java contendo um método implementado externamente

em tempo de execução a diferentes implementações da JVM (Stepanian 2005).

O acesso a atributos Java em código C é ilustrado através do seguinte exemplo. Inicialmente, na Figura 3.1, é implementada uma classe Java que possui um atributo privado, o array `elementos`, e define uma função, `segundoElemento`, a ser implementada em C[6].

A implementação de `segundoElemento` é mostrada na Fig. 3.2, mostrando a seqüência de chamadas até obter em C o elemento do array Java. Para acessar o atributo `elementos`, a função deve obter o identificador do campo. Para tal, devemos inicialmente obter uma referência da classe atual com `GetObjectClass` a partir da referência ao objeto (`this`) passada como parâmetro à função. De posse da referência da classe (`classe`), obtemos o identificador do campo com `GetFieldID`. O conteúdo do campo é então obtido com `GetObjectField`: uma referência para o array. Com este, o elemento do array é obtido com `GetObjectArrayElement`. Uma cópia do elemento, convertido para uma string C codificada em UTF-8, é retornada com `GetStringUTFChars`. Como discutido anteriormente, após o uso, a string deve ser liberada com `ReleaseStringUTFChars`.

A manipulação de objetos do tipo `Class` também é feita através de funções específicas. Não é possível criar classes Java através da API C, mas é possível carregar classes em tempo de execução usando a função `DefineClass`, que recebe um buffer contendo a representação de uma classe Java pré-compilada. Referências do tipo `jclass` podem ser obtidas através do nome da classe usando `FindClass`, que utiliza uma sintaxe de descritores de classe

[6]Os detalhes sobre a declaração e registro de funções implementadas em C serão discutidos na Seção 3.4.3.

```
#include <jni.h>
#include <stdio.h>
#include "ExemploJNI.h"

JNIEXPORT void JNICALL
Java_ExemploJNI_segundoElemento(JNIEnv* J, jobject this) {
   /* Obter a classe de this: ExemploJNI */
   jclass classe = (*J)->GetObjectClass(J, this);
   /* Obter o campo ExemploJNI.elementos, de tipo String[] */
   jfieldID elemsID = (*J)->GetFieldID(J, classe,
      "elementos", "[Ljava/lang/String;");
   /* Obter o conteúdo do campo ExemploJNI.elementos */
   jarray elems = (*J)->GetObjectField(J, this, elemsID);
   /* elems_1 = elementos[1] */
   jstring elems_1 = (*J)->GetObjectArrayElement(J, elems, 1);
   /* obter representação de elems_1 como uma string C */
   const char* elems_1_c = (*J)->GetStringUTFChars(J, elems_1, NULL);
   /* exibir a string */
   printf("%s\n", elems_1_c);
   /* liberar memória da string */
   (*J)->ReleaseStringUTFChars(J, elems_1, elems_1_c);
}
```

Figura 3.2: Código C implementando um método Java

similar à de descritor de campos usada por `GetFieldID`[7].

Para a atribuição de valores de C que podem ser convertidos para tipos primitivos de Java, a JNI possui funções como `SetIntField` e `SetFloatArrayRegion`. Para os demais tipos, não há uma provisão específica para o armazenamento de dados de C no espaço de objetos de Java. Nestes casos, a documentação sugere o armazenamento de ponteiros em tipos numéricos (Liang 1999), apesar dos problemas de portabilidade em que tal abordagem incorre.

3.1.4 Lua

A API de Lua define uma abordagem diferente para a manipulação de dados em C: não são expostos ao código C ponteiros ou *handles* para objetos Lua. As operações são definidas em termos de índices de uma pilha virtual. Assim, a transferência de dados de C para Lua se dá através de funções que recebem tipos de C, os convertem para valores Lua e os empilham, como `lua_pushboolean`, `lua_pushinteger` e `lua_pushlstring`. Diversas operações da API operam sobre o valor no topo da pilha, como por exemplo

[7]Tanto em descritores de classe como de campo, `"[Ljava/lang/String;"` representam `String[]`. Para o tipo `String`, entretanto, `"Ljava/lang/String;"` é o descritor de campo e `"java/lang/String"` o de classe.

`lua_setglobal`[8]:

```
lua_pushinteger(L, 123);   /* Insere o número 123 na pilha */
lua_setglobal(L, "foo");   /* Atribui o número 123 na global foo */
```

A maioria das funções de consulta, entretanto, permitem especificar um índice qualquer da pilha (com valores positivos para indexação a partir da base e negativos para índices a partir do topo).

A conversão de dados de Lua para C é feita através de funções como `lua_tonumber` e `lua_tolstring`, que recebem um índice da pilha, convertem o valor no índice para o tipo Lua especificado se necessário, e retornam o valor convertido para o tipo C equivalente. Números têm o tipo C `lua_Number`, que corresponde a `double` por padrão mas é um parâmetro na compilação de Lua. Strings, em particular, são objetos imutáveis e têm sua representação internalizada: duas strings de conteúdo idêntico compartilham a mesma representação interna.

Assim, diferentemente de linguagens como Python e Ruby, não é possível modificar o conteúdo de uma string Lua através de C via a sua representação em memória como um `char*`. Para tornar mais eficiente a construção em etapas de strings Lua a partir de C, a biblioteca auxiliar define um tipo C `luaL_Buffer` e funções como `luaL_addstring` e `luaL_addvalue`, que permitem a construção de uma string em etapas até que ela possa ser finalmente convertida para uma string Lua com `luaL_pushresult`. Assim, evitam-se consecutivas operações de concatenação de string através da API de Lua.

Lua define dois tipos de dados específicos para o armazenamento de dados para C, *full userdata* e *light userdata*. *Full userdata* descrevem blocos de memória gerenciados por Lua e utilizados por código C. Eles existem em Lua como objetos opacos, e são criados por `lua_newuserdata`, que insere o novo objeto na pilha de Lua e retorna a C um ponteiro com a área de memória do tamanho requisitado. Objetos do tipo *light userdata*, criados através de `lua_pushlightuserdata`, permitem armazenar ponteiros de C em Lua; a alocação e gerência do bloco de memória ficam a cargo do código C. O seguinte exemplo ilustra o uso de userdata, assumindo a mesma struct `point` definida na página 33. O userdata é criado da seguinte forma:

```
/* Cria o full userdata e o insere na pilha e retorna
   o ponteiro para C. A memória é alocada por Lua. */
point* full_p = (point*) lua_newuserdata(L, sizeof(point));
/* Utilizamos então o ponteiro em C... */
```

[8]Funções da API que operam sobre um estado de execução Lua recebem um parâmetro inicial (nos exemplos, chamado de `L`) indicando o estado a que se referem. Isto será discutido mais adiante, na Seção 3.4.4.

```
full_p->x = 100; full_p->y = 200;
/* Atribui o objeto à variável global Lua Ponto */
lua_setglobal(L, "Ponto");
```

A seguir, o acesso:

```
/* Empilha a global Ponto */
lua_getglobal(L, "Ponto");
/* Obtém o ponteiro C do userdata no topo da pilha (posição -1) */
point* p = (point*) lua_touserdata(L, -1);
printf("(%d,%d)\n", p->x, p->y);
/* Restaura a pilha à posição original, removendo o item.
   Ele não será coletado pois está associado à variável global. */
lua_pop(L, 1);
```

Assumindo que a global `Ponto` é a única referência para este bloco, para liberá-lo basta sobrescrevermos `Ponto`, por exemplo, com `nil`; a memória do full userdata poderá então ser liberada pelo coletor de lixo, como a de qualquer valor Lua sem referências.

```
lua_pushnil(L);
lua_setglobal(L, "Ponto");
```

A área de armazenamento da pilha não se ajusta dinamicamente e as funções da API não realizam controle de *overflow*. Assim, o programador é responsável por controlar o tamanho da pilha, através da função `lua_checkstack`. Na prática, o tamanho da pilha só irá crescer na ocorrência de laços empilhando elementos, já que seqüências típicas de operações tendem a empilhar valores e desempilhá-los em seguida.

Tabelas são o único tipo para construção de estruturas de dados em Lua. Lua oferece uma API completa para manipulação de tabelas a partir de C. Tabelas podem ser criadas com `lua_newtable` ou `lua_createtable`; a segunda forma permite pré-alocar memória para os elementos da tabela. As funções `lua_gettable` e `lua_settable` implementam a semântica de leitura e atribuição de campos em uma tabela em Lua, incluindo a possível chamada a metamétodos; para chamadas sem a invocação de metamétodos existem as funções `lua_rawget` e `lua_rawset`, equivalentes a `rawget` e `rawset` em Lua (além das variantes `lua_rawgeti` e `lua_rawseti` para conveniência). Há ainda a função `lua_next`, equivalente à função Lua `next`, que permite percorrer os elementos de uma tabela. Um exemplo de manipulação de tabelas é dado a seguir:

```
/* tabela["chave"] = 12345, em C: */
lua_getglobal(L, "tabela");
```

```
lua_pushstring(L, "chave");
lua_pushinteger(L, 12345);
/* lua_settable insere o item no topo da pilha
     na tabela especificada como argumento,
   usando como chave o elemento logo abaixo do topo: */
lua_settable(L, -3);
```

Diversos conceitos de Lua são representados através de tabelas – ambiente global, metatabelas, registro – e são assim tratados em C usando as funções da API para manipulação de tabelas. A tabela do ambiente global da *thread* em execução pode ser acessada através de um índice especial da pilha virtual, `LUA_GLOBALSINDEX`. Pode-se ainda definir uma tabela de ambiente da função, indexada em `LUA_ENVIRONINDEX`, para isolar dados a serem compartilhados internamente em funções de módulos escritos em C. Exemplificando, o ambiente global pode ser manipulado como uma tabela desta forma:

```
lua_pushstring(L, "Ponto");
lua_gettable(L, LUA_GLOBALSINDEX);
```

Isto é equivalente a:

```
lua_getglobal(L, "Ponto");
```

3.1.5 Perl

Os processos de estender e embutir Perl são bastante distintos. Para extensões, Perl oferece uma linguagem para descrição de interfaces chamada XS. Ao invés de isolar o acesso às estruturas internas de Perl através de uma API pública, a abordagem proposta é encapsular o processo de geração de código *wrapper* para a comunicação entre funções escritas em C e as estruturas internas de Perl usando interfaces escritas em XS. Arquivos `.xs` contêm código C acompanhado de anotações que simplificam o tratamento dos parâmetros de entrada e saída. Estes são alimentados ao pré-processador `xsubpp`, que geram então o código usando a API oferecida pela biblioteca de Perl. Esta biblioteca oferece acesso de baixo nível ao funcionamento do interpretador, permitindo, por exemplo, manipular o ponteiro da sua pilha interna. O objetivo de XS é ocultar ao desenvolvedor de extensões estes detalhes.

Para embutir o interpretador Perl em uma aplicação, a biblioteca que a implementa oferece algumas funções que permitem disparar um interpretador. Na API de mais alto nível, pode-se construir um array de parâmetros a serem passados para o interpretador de forma equivalente às opções do interpretador

Perl de linha de comando, inclusive usando a opção `"-e"` para executar trechos de código.

Os tipos de variáveis Perl são mapeados para structs em C: `SV` para escalares, `AV` para arrays, `HV` para hashes. Estes valores C são melhor entendidos como *containers* para valores Perl: uma variável escalar em Perl tem um `SV` associado a si; todavia, pode-se criar em C um `SV` que não esteja associado a nenhum nome de variável Perl.

Os *typedefs* `IV`, `UV`, `NV` e `PV` representam valores C correspondentes aos tipos primitivos de Perl, e correspondem, respectivamente, a inteiros com e sem sinal, valores de ponto flutuante e strings. Estes valores podem ser copiados para `SV`s. Referências Perl são representadas como `RV`, e também são um tipo de `SV`. Há ainda o tipo `GV`, capaz de referenciar qualquer tipo representável em uma variável Perl.

Variáveis do espaço de Perl são acessadas com `get_sv`, `get_av` e `get_hv`. Estas funções recebem uma string C com o nome da variável (possivelmente qualificado da forma `"pacote::variavel"`). O conteúdo de valores escalares é convertido de volta para um tipo C com as macros `Sv*`: `SvIV` retorna um inteiro, `SvPV` retorna um `char*` e o comprimento da string no segundo parâmetro, etc. O seguinte código C exibe o conteúdo da variável Perl `$a`, assumindo que ela contenha um valor inteiro:

```
printf("a = %d\n", SvIV(get_sv("a", FALSE)));
```

A flag passada como segundo parâmetro de `get_sv/av/hv` indica se a variável deve ser criada se o nome passado não corresponder a uma variável existente. Passar um nome inexistente e usar `TRUE` no segundo parâmetro é uma forma conveniente de criar uma nova variável acessível no espaço de C já realizando o *binding* desta no espaço de Perl.

```
/* Cria uma variável do tipo array,
   acessível em Perl como a global @arr e em C como o AV* arr */
AV* arr = get_av("arr", TRUE);
```

Um `SV` pode ser criado em C com as funções `newSV*`: `newSViv` gera um novo `SV` armazenando um inteiro com sinal; `newSVpv` a partir de uma string, e assim por diante. A função `newSV` permite criar um `SV` com área de memória não inicializada, acessível através da função `SvPVX`, permitindo assim criar escalares com valores arbitrários gerenciados por código C. Usando o mesmo exemplo da struct `point` das seções anteriores, podemos armazenar um objeto C em um valor Perl da seguinte forma:

```
/* Aloca um SV não inicializado do tamanho de um point */
```

```
SV* v = newSV(sizeof(point));
/* Obtém o ponteiro para a área de memória do SV */
point* p = (point*) SvPVX(v);
/* Manipula o point em C. Quando v for retornado para Perl,
ele será uma variável opaca (seu conteúdo não será acessível
via Perl). */
p->x = 100; p->y = 200;
```

Valores são atribuídos a SVs usando as funções sv_set*: sv_setiv, sv_setpv, etc. As funções para manipulação de strings possuem variantes como newSVpvn e sv_setpvf, que permitem especificar o comprimento da string ou realizar formatação como em sprintf. Para strings, há ainda funções sv_cat*, que atuam como sv_set* mas concatenam o valor dado ao conteúdo atual da string ao invés de substituí-lo. A função sv_setsv copia o valor de um SV para outro. O SV criado no exemplo anterior pode ser atribuído para uma variável global da seguinte forma:

```
/* Obtém o SV da variável global $ponto */
SV* ponto = get_sv("ponto", TRUE);
/* Atribui o valor de v para ponto */
sv_setsv(ponto, v);
```

O tipo dos dados armazenados em SVs é verificado com as macros SvIOK para inteiros, SvNOK para valores de ponto flutuante e SvPOK para strings. Estas funções retornam sucesso se o escalar é conversível para o tipo especificado – as variantes SvIOKp, SvNOKp, SvPOKp verificam se o valor armazenado no SV é realmente do tipo.

Arrays e hashes são criados com newAV e newHV. Arrays podem ser populados com um array C de ponteiros para SV através de av_make. Operações como av_fetch, av_pop, hv_fetch e hv_exists permitem operar sobre elementos das estruturas. Em av_fetch e hv_fetch, o tipo de retorno é SV**, para diferenciar o retorno de um elemento existente que aponta para NULL de um elemento não encontrado. No exemplo a seguir, criaremos um array Perl contendo os 10 primeiros elementos da série de Fibonacci:

```
/* Cria um novo array. */
AV* a = newAV();
/* Armazena dois valores, 0 e 1, nas primeiras posições do array. */
av_push(a, newSViv(0));
av_push(a, newSViv(1));
for (int i = 2; i < 10; i++) {
   /* Obviamente seria mais eficiente armazenar os valores em
      variáveis temporárias em C, mas obtenhamos os dois últimos
      valores da seqüência de volta do array Perl para fins de
```

```
        ilustração: */
    SV** penultimo_sv = av_fetch(a, i-2, FALSE);
    SV** ultimo_sv = av_fetch(a, i-1, FALSE);
    /* Obtém os inteiros armazenados nos SVs */
    int penultimo = SvIV(*penultimo_sv);
    int ultimo = SvIV(*ultimo_sv);
    /* Cria um novo SV e o insere no final do array */
    av_push(a, newSViv( penultimo + ultimo ) );
}
```

Uma vez criado este `AV`, entretanto, não há uma forma de associá-lo a uma variável Perl. Seu conteúdo deve ser copiado item a item. Para que este seja acessível a partir de Perl, deveríamos tê-lo criado com `get_av`, e não `newAV`. A utilidade de `AV`s não associados a variáveis está na passagem de parâmetros na chamada de funções e como valores de retorno.

Algumas funções para manipulação de hashes expõem os pares chave/valor como ponteiros `HE`. As macros `HeSVKEY` e `HeVAL` permitem extrair a chave e valor de um `HE`. A seguinte função exibe em C os elementos de uma hash Perl:

```
void imprime_hash(HV* hash) {
    HE* item;
    /* Cada HV mantém o seu controle interno de iteração */
    hv_iterinit(hash);
    /* Obtém o próximo par chave/valor da iteração */
    while ( (item = hv_iternext(hash)) ) {
        /* Obtém a representação string dos escalares
           representando chave e valor do item */
        char* chave = SvPV_nolen(HeSVKEY(item));
        char* valor = SvPV_nolen(HeVAL(item));
        printf("%s => %s\n", chave, valor);
    }
}
```

Cuidados especiais devem ser tomados ao utilizar os valores `undef`, `true` e `false` em arrays e hashes, embora Perl exponha estas constantes na API de C (`PL_sv_undef`, `PL_sv_true`, `PL_sv_false`). A constante `PL_sv_undef` é usada internamente na implementação de `AV`s e `HV`s, e a atualização de valores em `HV`s ocorre *in-place*, o que gera problemas ao atualizar elementos contendo estas constantes. A documentação recomenda gerar cópias destes valores ao usá-los em estruturas `AV` e `HV` (Okamoto 2006b).

Referências Perl são criadas com `newRV_inc` e `newRV_noinc`, que recebem um ponteiro para `SV`, `AV` ou `HV` como parâmetro (as duas funções diferem entre si no que tange à contagem de referências, que será abordada na Seção 3.2.5). O

valor apontado por uma referência é obtido com `SvRV`. O retorno desta macro deve ser convertido via *cast* para o tipo apropriado (`IV`, `PV`, `AV`, etc.), que pode ser verificado com `SvTYPE`.

Diversas funções da API têm tipo de parâmetros ou retorno declarados como `SV` quando na verdade aceitam `AV`s ou `HV`s; isto é análogo ao conceito de *contextos* de Perl, onde um mesmo valor pode ser tratado como lista (array ou hash) ou escalar dependendo da expressão onde ele é inserido (Marquess 2006). Em código Perl, o contexto em que uma função está executando pode ser verificado com `wantarray`. Em C, o contexto pode ser verificado com a macro `GIMME_V`, que retorna `G_VOID`, `G_SCALAR` ou `G_ARRAY`.

3.1.6 Comparação

O conjunto básico de funções para manipulação de dados nas cinco linguagens discutidas é similar: todas elas possuem funções para converter valores da linguagem para tipos básicos de C e vice-versa. Todas oferecem ainda funções para a manipulação dos seus tipos estruturados fundamentais (tabelas em Lua, arrays em Java, arrays e hashes em Ruby e Perl, listas e dicionários em Python). Python, em particular, define uma API extensa de funções para operações sobre as suas classes *built-in*; a maioria destas operações poderia ser realizada usando a API genérica de invocação de métodos, mas são oferecidas diretamente em C como conveniência.

Lua ganha destaque por possuir, com o seu modelo de pilha, a API para manipulação de dados mais simples e ortogonal dentre as estudadas. Entretanto, muitas vezes o código resultante perde em legibilidade quando os índices da pilha são pouco óbvios. É comum ver código C usando a API de Lua comentado linha a linha, para evitar que o programador tenha que simular mentalmente o funcionamento da pilha ao ler o programa.

Em Java, a tipagem estática reduz muito a necessidade de conversões de dados explícitas no código C. Por outro lado, o tratamento de *multi-threading* complica o acesso a tipos como strings e arrays.

Um ponto negativo na API de Ruby é a exposição de detalhes de implementação dos campos da struct que descreve o seu tipo fundamental `VALUE`. Isto restringe a flexibilidade da implementação da linguagem e é uma prática de programação pouco segura. Perl também expõe grande parte de suas estruturas internas; não de forma tão direta quanto Ruby, mas através de macros. Tais macros, entretanto, assumem a aderência a protocolos tão estritos de uso que na prática também limitam largamente as possiblidades de alterações na implementação (um exemplo é a seqüência para chamada de

funções, que será apresentada na Seção 3.3.5).

A criação de dados que contêm estruturas C armazenados pela linguagem de script é uma tarefa fácil em Perl, Ruby e Lua: Perl permite criar `SV`s contendo blocos de memória arbitrários para uso em C; Ruby disponibiliza a macro `Data_Wrap_Struct` que gera um objeto Ruby que encapsula uma estrutura C; Lua define um tipo básico na linguagem especialmente para este fim. Já em Python, o processo é trabalhoso. Criar uma classe Python a partir de C envolve declarar partes dela estaticamente e outras partes dinamicamente, sendo usualmente necessário definir três estruturas C diferentes. Em Java, não é possível criar novos tipos a partir de C, apenas carregar classes.

Outra tarefa comum ao interagir com C é a necessidade de armazenar ponteiros no espaço de dados da linguagem de script. Python, Lua e Perl oferecem recursos para fazer isto de forma direta: criando um objeto `PyCObject` em Python; um light userdata em Lua; ou armazenando o ponteiro na área de dados de um `SV` em Perl. Em Java e Ruby, a alternativa é converter os ponteiros e armazená-los como números. De fato, isto ocorre internamente na implementação de Ruby, e as limitações de portabilidade desta abordagem são evidenciados pelo fato de que a compilação de Ruby falha se `sizeof(void*) != sizeof(long)`.

Outro aspecto que merece nota é a preocupação em não poluir o espaço de nomes de C. Python, Java e Lua definem todas as suas funções e tipos C com prefixos que visam evitar conflitos com nomes definidos pela aplicação. Já Perl e Ruby definem nomes de forma desorganizada, o que ocasionalmente causa problemas[9]. Perl possui opções para desabilitar uma série de macros e forçar um prefixo comum em suas funções, mas este recurso é incompleto e usá-lo prejudica a funcionalidade dos seus cabeçalhos[10].

3.2 Coleta de lixo

A partir do momento em que código C ganha acesso a referências a dados armazenados no espaço de armazenamento de outra linguagem, sejam ponteiros ou identificadores, o programador deve levar em consideração as diferenças entre os modelos de gerência de memória envolvidos, já que código executado na outra linguagem pode liberar o dado. Por exemplo, o programa C pode desalocar o objeto referenciado em um dado da linguagem de script,

[9]Por exemplo, conflitos deste tipo ocorreram nos bindings Ruby do sistema de controle de versão Subversion em plataformas Win32 (`http://svn.haxx.se/dev/archive-2005-04/1789.shtml`).

[10]No estudo de caso apresentado no Capítulo 4, ao utilizar a API de Perl procuramos utilizar somente as versões com prefixo `Perl_` das funções da API, mas diversas macros só estão disponíveis nas versões sem o prefixo.

ou a linguagem de script pode remover um elemento de uma estrutura fazendo com que ele seja coletado. Em princípio, esta tarefa de manter a consistência entre os dois ambientes não é diferente da gerência de memória realizada normalmente pelo programador em C. Entretanto, a interação com algumas linguagens adiciona um complicador: os mecanismos de coleta de lixo realizam liberação de dados da memória de forma implícita. O princípio fundamental da coleta de lixo dita que um objeto não é coletado caso haja algum elemento (variável, estrutura de dados) apontando para ele. Todavia, o mesmo não vale para o ambiente C: a presença de um ponteiro apontando para um objeto não garante que ele não será coletado, uma vez que o coletor de lixo não gerencia os ponteiros do código C.

É preciso, então, indicar a partir do código C que os dados que continuam acessíveis por ele não devem ser coletados. De forma complementar, ao transferir o controle de objetos C para o domínio da outra linguagem – por exemplo, para armazená-los em uma estrutura de dados desta – é necessário indicar à linguagem como desalocar a memória da estrutura quando o coletor de lixo detectar que ela não está mais em uso. A forma como a API irá fornecer estas funcionalidades depende não só do projeto da API para C, mas também do modelo de coleta de lixo empregado pela implementação da linguagem.

3.2.1 Python

A máquina virtual de Python possui um coletor de lixo baseado em contagem de referências. Como a API de Python retorna ao código C ponteiros a `PyObject`s, o programador deve ter o cuidado de garantir que eles se mantenham válidos. Para isto, é necessário incrementar e decrementar o contador de referências do objeto apontado conforme deseja-se manter a validade dos ponteiros em código C.

De maneira geral, uma vez que código C deseja reter um `PyObject*`, ele deve utilizar a macro `Py_INCREF` para incrementar a sua contagem de referências e assim impedi-lo de ser coletado. Uma vez que o valor não seja mais necessário, decrementa-se a contagem similarmente com `Py_DECREF`. Python trabalha com o conceito de *"propriedade de referências"* para definir quando o programador deve incrementar ou decrementar o contador de referências retornadas pelas funções da API. A maior parte das funções da API que retornam ponteiros a `PyObject`s *transferem* referências para o chamador; a referência passa então a ser sua responsabilidade – ele pode passá-la adiante ou terá o dever de decrementá-la com `Py_DECREF` assim que não precisar mais usá-la (o código C pode guardar referências que sejam de sua propriedade em suas estru-

```
void bug(PyObject* list) {
   PyObject* item = PyList_GetItem(list, 0);
   if (!item) return;
   PyList_SetItem(list, 1, PyInt_FromLong(0L));
   PyObject_Print(item, stdout, 0); /* BUG! */
}
```

(a) Acesso possivelmente inválido em `PyObject_Print`

```
void no_bug(PyObject* list) {
   PyObject* item = PyList_GetItem(list, 0);
   if (!item) return;
   Py_INCREF(item);
   PyList_SetItem(list, 1, PyInt_FromLong(0L));
   PyObject_Print(item, stdout, 0);
   Py_DECREF(item);
}
```

(b) `item` é garantidamente válido em `PyObject_Print`

Figura 3.3: Acesso possivelmente inválido a uma referência a um `PyObject` em código C

turas de dados; elas continuarão válidas mesmo após o retorno da função, até serem explicitamente decrementadas). Outras funções *emprestam* referências; o código que recebe uma referência deste tipo não precisa decrementá-la ao terminar de utilizá-la, mas a validade do objeto está atrelada à validade da referência no objeto que a retornou ao código C. Por exemplo, `PyList_GetItem` empresta uma referência a um elemento da lista. O ponteiro retornado continuará válido enquanto este item continuar contido na lista. Pode-se obter a "propriedade" de uma referência emprestada incrementando a contagem do objeto com `Py_INCREF`: a validade do ponteiro passa a ser independente do objeto *container* que o retornou, mas o código C passa a ser responsável por decrementar a referência posteriormente com `Py_DECREF`.

Para referências de objetos passadas de código C de volta para Python, há dois casos na API em que funções *"roubam"* referências, isto é, em que a referência deixa de pertencer à função C chamadora: `PyList_SetItem` e `PyTuple_SetItem`. A referência passada, que pertencia ao chamador, passa então a pertencer à lista ou tupla. No contexto do chamador, ele passa a ser uma referência emprestada, que não precisa mais ser decrementada. O objetivo disto é permitir chamadas de função aninhadas onde, por exemplo, o parâmetro de `PyList_SetItem` é uma chamada que gera um novo objeto a ser armazenado na lista. Assim, evita-se que o programador tenha que armazenar um ponteiro para o objeto apenas para decrementar a sua referência posteriormente.

A interação com o contador de referências pode ser bas-

tante sutil. O exemplo da Figura 3.3, extraído da documentação de Python (van Rossum 2006a), demonstra que uma referência pode ser invalidada por código aparentemente não relacionado[11]. À primeira vista, a inclusão de um elemento em `list[1]` parece não afetar a referência `item`, que corresponde a `list[0]`. Todavia, a inclusão de `list[1]` pode haver removido da lista um elemento que se encontrava nesta posição. Caso a lista fosse a última referência válida para o elemento, este poderia ser coletado. A coleta do objeto pode invocar o seu método finalizador `__del__`, que pode rodar código Python arbitrário. Se este código remove o elemento da posição 0 de `list` e isto causar a sua coleta, a referência `item` passa a ser inválida, porque `PyList_GetItem` retorna uma referência emprestada.

Ao implementar funções em C que retornam referências a `PyObject`s, o mesmo cuidado de definir a política de tempo de vida da referência deve ser tomado. Para retornar uma referência nova, pode ser necessário incrementar a contagem do objeto. Isto se manifesta, por exemplo, na forma correta de uma função C retornar o valor `None`, que envolve chamar `Py_INCREF(Py_none)`[12]. Mesmo objetos Python representando números devem ter a sua contagem de referências controlada pelo programador C.

Para que o código C possa realizar operações de finalização sobre os dados armazenados em um tipo Python definido em C, é possível definir uma função de desalocação no campo `tp_dealloc` da estrutura `PyTypeObject` que descreve o tipo. Esta função é normalmente responsável por liberar recursos alocados via código C (arquivos abertos, ponteiros para áreas de memória não acessíveis por Python, etc.) e decrementar referências a outros objetos Python mantidas pelo objeto.

Ao desalocar estruturas de dados como listas pode-se provocar uma cadeia arbitrariamente grande de desalocações, à medida que cada elemento desencadeia a desalocação do próximo elemento da estrutura. Isto dispara a função de desalocação recursivamente e poderia facilmente causar estouro da pilha de C. Para contornar este problema, Python inclui um par de macros, `Py_TRASHCAN_SAFE_BEGIN` e `Py_TRASHCAN_SAFE_END`, que controla o número de níveis de recursão aceitos. A cada execução de `Py_TRASHCAN_SAFE_BEGIN` um contador interno é incrementado. Enquanto este contador não atingir o valor limite definido na constante `PyTrash_UNWIND_LEVEL` (50 por padrão), a função executa normalmente. Quando o limite é atingido, `Py_TRASHCAN_SAFE_BEGIN` armazena o objeto

[11] De fato, a documentação informa que versões antigas de Python continham variantes deste *bug* em alguns de seus módulos.

[12] Este padrão é tão comum que a seqüência `Py_INCREF(Py_none); return Py_none;` foi encapsulada na macro `Py_RETURN_NONE`.

em uma lista interna e salta diretamente para `Py_TRASHCAN_SAFE_END`, evitando desalocar o objeto e entrar em recursão novamente. Ao final de cada nível da recursão, a macro `Py_TRASHCAN_SAFE_END` decrementa o contador. Quando o contador chega a zero, `Py_TRASHCAN_SAFE_END` dispara novamente `tp_dealloc` sobre os elementos armazenados na lista interna, reiniciando assim a recursão sobre a estrutura. Assim, uma cadeia de n desalocações é quebrada em n/`PyTrash_UNWIND_LEVEL` cadeias, nenhuma excedendo `PyTrash_UNWIND_LEVEL` níveis de recursão na pilha de C. As implementações dos principais tipos estruturados de Python, como listas, tuplas e dicionários, fazem uso deste mecanismo.

O modelo de coleta de lixo utilizando contagem de referências traz consigo preocupações sobre referências circulares: uma cadeia de objetos que mantêm referências entre si mantém a contagem de cada um dos seus elementos acima de zero, mesmo que eles não sejam alcançáveis a partir de nenhum outro objeto. Python inclui um detector de ciclos, mas cuidados especiais devem ser tomadas para que tipos implementados em C se comportem corretamente caso possam gerar ciclos. Deve-se implementar uma função para percorrer referências contidas no objeto e uma função que decremente a contagem destas referências. Estas funções devem ser registradas nos campos `tp_traverse` e `tp_clear` da estrutura `PyTypeObject`. A função `tp_clear` deve ter o cuidado de zerar o valor de seus campos `PyObject*` para `NULL` antes de decrementar cada referência, uma vez que a operação de decremento pode iniciar a desalocação do objeto e disparar uma chamada a `tp_traverse` que, devido ao ciclo, retorne ao objeto anterior. O tipo deve ser, então, identificado com a flag `Py_TPFLAGS_HAVE_GC` no campo `tp_flags` de `PyTypeObject`.

Além disso, a implementação de objetos Python que suportem coleta cíclica em C implica ainda em outros cuidados. Objetos devem ser alocados com `PyObject_GC_New` ou `PyObject_GC_NewVar` ao invés da funções usuais `PyObject_New` e `PyObject_NewVar`. Durante a construção do objeto, após os campos a serem visitados por `tp_traverse` terem sido preenchidos, é necessário ainda chamar uma função de notificação `PyObject_GC_Track` e durante a desalocação, antes de invalidar os campos do objeto, chamar `PyObject_GC_UnTrack`. Para objetos que necessitem do mecanismo de *"trashcan"* para evitar estouro de pilha, é ainda preciso tomar o cuidado de desmarcar o objeto com `PyObject_GC_UnTrack` antes de entrar no bloco `Py_TRASHCAN_SAFE_BEGIN/END`.

Apesar de oferecer um mecanismo de detecção de ciclos, Python é incapaz de coletar ciclos cujos objetos contenham finalizadores implementados em Python (métodos `__del__`); a única forma de acessar estes objetos é então

através da lista `garbage` no módulo `gc`. Este módulo (acessível a partir de C através de chamadas de função Python via API) oferece uma interface com o coletor de lixo, incluindo funções como `enable` e `disable`, para ativar e desativar o coletor de lixo; `collect`, para executar uma coleta; `get_objects`, que retorna uma lista contendo todos os objetos controlados pelo coletor (exceto a própria lista); `get_referrers` e `get_referents`, que retornam a lista de objetos que referenciam ou são referenciados por um dado objeto – estas listas são obtidas percorrendo objetos com a função `tp_traverse`, o que pode não apontar para todos os objetos realmente alcançáveis, ou ainda retornar objetos em estado inválido (como objetos em ciclos ainda não coletados ou objetos ainda não totalmente construídos) e assim devem ser usadas apenas para fins de depuração.

3.2.2 Ruby

Ruby utiliza um coletor de lixo *mark-and-sweep* (Wilson 1992). Esta técnica evita o problema de referências cíclicas enfrentado por Python, bastando que os objetos válidos possam ser corretamente indicados como alcançáveis.

Objetos alcançáveis a partir do espaço de objetos de Ruby – atribuídos a uma variável global Ruby ou inseridos em alguma estrutura de dados alcançável em Ruby – não estarão sujeitos à coleta de lixo. Além destes, temos ainda os objetos retornados por Ruby para o espaço de C, já que muitas funções de API retornam `VALUE`s. A documentação alerta que, para armazenar objetos Ruby em C, seja em variáveis globais ou estruturas de dados, é necessário notificar à máquina virtual que o `VALUE` não deve ser coletado usando a função `rb_global_variable` (Thomas 2004) (Embora a documentação não informe, é possível desmarcar um valor global com `rb_gc_unregister_address`).

Objetos de escopo local a uma função C, entretanto, não precisam ser notificados. A forma como Ruby garante a validade de `VALUE`s locais é bastante peculiar: ao realizar a fase de marcação, o coletor de lixo varre a pilha de C em busca de valores que se pareçam com endereços de `VALUE`s, isto é, seqüências numéricas que correspondam a endereços de `VALUE`s válidos. Estes endereços podem ser identificados pois objetos são sempre alocados dentro de *heaps* mantidas pelo interpretador Ruby. Cada `VALUE` encontrado na pilha é então marcado. Isto garante que nenhum `VALUE` localmente alcançável por código C seja invalidado, mas pode gerar "falsos positivos" evitando que dados que poderiam ser coletados o sejam.

Apesar da conveniência para o programador, tal abordagem é extrema-

mente não-portável. A implementação do coletor de lixo em Ruby 1.8.2 possui `#ifdef`s para IA-64, DJGPP, FreeBSD, Win32, Cygwin, GCC, Atari ST, AIX, MS-DOS, Human68k, Windows CE, SPARC e Motorola 68000. Além disso, o coletor força a descarga dos registradores para a pilha usando `setjmp`, para evitar que variáveis do tipo `VALUE` que tenham sido otimizadas pelo compilador deixem de ser verificadas.

Conforme visto na Seção 3.1.2, objetos Ruby criados com `Data_Wrap_Struct` contêm structs C, que podem conter referências a `VALUE`s Ruby. A struct encapsulada, entretanto, é opaca para o universo Ruby. Então, para garantir que estes `VALUE`s sejam marcados como alcançáveis durante a coleta de lixo é necessário fazer isto via código C. `Data_Wrap_Struct` recebe, além da struct a encapsular, dois ponteiros, um para uma função de marcação e outra para uma função de desalocação. Quando o coletor de lixo visita o objeto na fase de marcação, ele invoca a função registrada, que deve chamar `rb_gc_mark` em cada um dos `VALUE`s armazenados na *struct* do objeto, informando assim que os objetos são alcançáveis. Quando um objeto encapsulado via `Data_*_Struct` for dado como não alcançável, a função de desalocação registrada é chamada. Para estruturas que não armazenam outros `VALUE`s, pode-se definir a função de marcação como `NULL` e a de desalocação como `free`.

Ruby possui um módulo `GC` que disponibiliza funções para ativar e desativar o coletor (`GC.enable` e `GC.disable`), bem como disparar uma coleta imediatamente (`GC.start`). Na API em C há funções equivalentes: `rb_gc_enable`, `rb_gc_disable` e `rb_gc_start`. A API de C inclui ainda uma função que insere um objeto imediatamente na lista de objetos a serem reciclados pelo alocador de memória de Ruby, `rb_gc_force_recycle`. Esta função deve ser usada com cuidado, já que se houverem outras referências apontando ao objeto elas passarão a apontar para outro objeto quando a área de memória for reutilizada pelo alocador de Ruby.

Ruby oferece ainda como conveniência para o programador C alguns *wrappers* para as funções `malloc` e `realloc` que interagem com o coletor de lixo, forçando a execução do coletor durante alocações grandes[13] ou em situações de pouca memória disponível.

3.2.3

[13]O limite para definir "grande" se ajusta de acordo com o funcionamento do coletor e com as alocações realizadas anteriormente.

Java

Assim como Python e Ruby, a API de Java retorna referências a objetos da máquina virtual que podem ser armazenados em variáveis C. A JNI define três tipos de referências, *locais*, *globais* e *globais fracas*, para auxiliar no controle do tempo de vida destas e a sua interação com o coletor de lixo.

Referências locais são retornadas pela maioria das funções da JNI e são válidas até o retorno da função C que a obteve. Não é necessário desalocar explicitamente uma referência local: durante a execução de uma função C, a JVM mantém uma lista de referências locais passadas à função e libera todas elas quando o controle retorna à máquina virtual. Isto faz com que, de maneira geral, o programador não precise se preocupar com o coletor de lixo enquanto manipula valores retornados durante uma função. Por outro lado, em código que pode utilizar uma grande quantidade de referências locais é mais eficiente desalocar referências locais explicitamente, usando `DeleteLocalRef`. A partir da versão 1.2 de Java, funções foram adicionadas para permitir a gerência de referências locais em blocos. `PushLocalFrame` e `PopLocalFrame` permitem criar escopos aninhados de referências locais, que são desalocados de uma só vez. `PushLocalFrame` recebe ainda um parâmetro indicando um número de *slots* a serem pré-alocados, como otimização. Esse valor pode ser configurado também com `EnsureLocalCapacity`.

Referências globais são geradas a partir de referências locais usando `NewGlobalRef`. Referências deste tipo mantêm-se válidas até serem explicitamente desalocadas com `DeleteGlobalRef`. Uma referência global impede o objeto de ser coletado, podendo assim, ser utilizada para armazenar objetos Java no espaço de C além da duração de uma função, por exemplo, em variáveis globais ou estáticas.

A Figura 3.4 mostra um exemplo do tipo de gerência de referências necessário quando se tem um laço criando referências temporárias sobre um número arbitrário de objetos.

No exemplo, a função `Java_Exemplo_concatArray` (equivalente, portanto, ao método `concatArray` de uma classe `Exemplo`) converte os elementos de um array para strings usando `Object.toString` e os concatena usando `String.concat`. Note que, como o número de vezes que o laço executa depende do tamanho do array passado, deve-se evitar que o número de referências cresça em cada iteração. Para isso, as opções seriam ou usar `Push/PopLocalFrame`, ou destruir as referências uma a uma com `DeleteLocalRef`. Se usássemos `Push/PopLocalFrame` no exemplo, teríamos que manter a string concatenada até o momento em uma referência global. Além disso, esta referência teria que ser destruída e recriada a cada iteração, já que strings são imutáveis em Java.

```
static jmethodID concat = NULL, toString = NULL;
/* Fazer o caching de jmethodIDs em código C é uma técnica comum.
   Vale lembrar que jmethodIDs não são objetos Java, e portanto
   não estão sujeitos a coleta de lixo. */
void cache_ids(JNIEnv* J) {
  jclass cls = (*J)->FindClass(J, "java/lang/String");
  concat = (*J)->GetMethodID(J, cls, "concat",
     "(Ljava/lang/String;)Ljava/lang/String;");
  cls = (*J)->FindClass(J, "java/lang/Object");
  toString = (*J)->GetMethodID(J, cls, "toString",
     "()Ljava/lang/String;");
}

JNIEXPORT jstring JNICALL
Java_Exemplo_concatArray(JNIEnv* J, jobject this, jobjectArray a) {
   if (!concat) cache_ids(J);
   /* s = "" */
   jstring s = (*J)->NewString(J, NULL, 0);
   /* len = a.length */
   int len = (*J)->GetArrayLength(J, a);
   for (int i = 0; i < len; i++) {
      /* o = a[i] */
      jobject o = (*J)->GetObjectArrayElement(J, a, i);
      /* os = o.toString() */
      jstring os = (*J)->CallObjectMethod(J, o, toString);
      /* s2 = s.concat(os) */
      jstring s2 = (*J)->CallObjectMethod(J, s, concat, os);
      (*J)->DeleteLocalRef(J, s);
      (*J)->DeleteLocalRef(J, o);
      (*J)->DeleteLocalRef(J, os);
      s = s2;
   }
   return s;
}
```

Figura 3.4: Rotina para concatenar os elementos de um array representados como strings.

Como o número de locais é pequeno, é mais conveniente neste caso controlá-las explicitamente com `DeleteLocalRef` do que recorrer a referências globais.

`PopLocalFrame` permite, através de um parâmetro adicional, transferir uma referência local do conjunto que está sendo desempilhado para o escopo exterior de referências locais, criando assim uma nova referência. Para o exemplo da Figura 3.4, isto ainda não evitaria a necessidade de liberar referências locais explicitamente a cada iteração do laço, já que cada `PopLocalFrame` criaria uma referência local nova.

A partir da versão 1.2, a JNI inclui referências globais fracas, com o objetivo de oferecer uma forma simplificada das referências fracas de Java (`java.lang.ref`) – um objeto que esteja sendo apontado apenas por referências globais fracas pode ser coletado. Originalmente, a API incluiu a função `IsSameObject` como forma de verificar a validade de uma referência

fraca, mas evidentemente este método é insuficiente: como Java é *multi-threaded*, o coletor de lixo pode invalidar a referência entre o teste e a instrução seguinte no código C. A documentação revisada avisa sobre esta limitação e recomenda o uso de referências globais, além de alertar sobre comportamentos indefinidos nas relações entre referências globais fracas em C e os tipos de referências fracas definidos em Java (Sun 2003).

Além desta, outras questões surgem a partir da combinação do modelo *multi-threaded* de Java com a exposição de referências a objetos da máquina virtual ao código C. Para reduzir o volume de cópia de dados entre Java e C, a JNI oferece algumas funções que retornam e liberam ponteiros para a representação interna de strings e arrays de tipos primitivos: `Get/ReleaseStringCritical` e `Get/ReleasePrimitiveArrayCritical`. O uso destas funções, porém, possui importantes restrições. A API especifica que, uma vez obtido um ponteiro através destas funções, o código C não deve chamar outras funções da JNI ou realizar chamadas que possam bloquear a thread atual e fazer com que ela espere por outra thread Java, sob o risco de causar um *deadlock*. Recomenda-se não manter blocos de memória bloqueados usando estas funções por muito tempo já que uma das técnicas possíveis para a implementação desta "região crítica" consiste em desabilitar o coletor de lixo da JVM. É importante notar ainda que referências locais e o ponteiro para o ambiente JNI passado para funções nativas são válidos apenas na thread onde foram criados; referências globais podem ser compartilhadas entre threads.

Além do mecanismo de referências fracas fornecido pelas classes do pacote `java.lang.ref`, a única forma oferecida por Java para interagir de maneira mais direta com o coletor de lixo é através da chamada `System.gc()`, que solicita à máquina virtual que dispare tão logo quanto possível a thread de coleta para que esta desaloque objetos não alcançáveis. Não há uma função C equivalente na JNI, mas este método pode ser invocado a partir de C com `CallStaticVoidMethod`.

3.2.4 Lua

Por não retornar referências explícitas a objetos Lua ao código C, a interação do código nativo com o coletor de lixo é bastante simplificada. Operações sobre objetos Lua são sempre especificadas através de índices da pilha virtual. Assim, a máquina virtual mantém o controle sobre quais os objetos são acessíveis a partir de C em qualquer dado momento.

Embora ponteiros para objetos não sejam manipulados na API, algumas funções retornam ponteiros para estruturas gerenciadas por Lua:

`lua_newuserdata`, `lua_to*string` e `lua_touserdata`. A validade dos ponteiros retornados por estas funções é dependente do tempo de vida do objeto correspondente a eles; para strings em particular, o ponteiro retornado só é garantidamente válido enquanto a string estiver na pilha. Lua oferece ainda a função `lua_topointer`, que permite obter ponteiros para alguns tipos de objetos (*userdata*, tabelas, *threads* e funções), mas somente com o objetivo de obter informação para depuração, já que não é possível converter o ponteiro de volta para um valor Lua.

O conteúdo da pilha virtual é zerado quando a função C retorna o controle à máquina virtual de Lua. Dessa forma, não é possível reter ponteiros retornados por Lua para uso posterior em variáveis globais ou estruturas de C. Por outro lado, a API oferece um mecanismo para armazenar valores Lua em uma localização conhecida por código C que não pode ser alterada via código Lua: o *registro*. O registro é uma tabela disponibilizada pela API de Lua para o armazenamento de valores Lua a partir de C; esta tabela não é normalmente acessível a partir de Lua. Como a tabela que implementa o registro é parte do *root set* do coletor de lixo, a inclusão de um objeto nesta tabela o impede de ser coletado, mantendo-o no registro até que seja explicitamente removido via código C.

Usando o registro, uma forma possível de descrever dados do espaço de Lua em estruturas de dados em C é armazenar os dados no registro e armazenar os índices usados na estrutura C. A biblioteca auxiliar de Lua encapsula tal idioma através de duas funções, `luaL_ref` e `luaL_unref`. A função `luaL_ref` associa o valor Lua passado a uma chave numérica inteira no registro, e retorna este número. Este valor então ser visto como um *handle* de alto nível para o objeto: o código C pode armazená-lo em variáveis ou estruturas e utilizá-lo para referenciar o objeto através do campo no registro. A função `luaL_unref` remove o valor Lua do registro e libera o índice para reuso. Para o bom funcionamento deste mecanismo, chaves inteiras não devem ser utilizadas diretamente pelo programador para armazenar dados no registro.

A API permite associar a objetos do tipo *full userdata* uma função de desalocação, `__gc`, na sua metatabela. Quando presente, esta função será tipicamente implementada em C, realizando a finalização de recursos. Por exemplo, o metamétodo `__gc` de objetos retornados pela função Lua `io.open` é uma função C que fecha o descritor de arquivo correspondente com a função C `fclose`.

A princípio, o fato de que é possível obter e modificar a metatabela de um userdata via código Lua pode parecer problemático, já que pode-se substituir o seu finalizador em `__gc`. Entretanto, funções de coleta implementadas em C

tipicamente validam o userdata recebido verificando o seu "tipo", identificado através da metatabela. Assim, mesmo que código Lua manipule a metatabela, uma função de coleta implementada em C que use `luaL_checkudata` não será levada a operar sobre um *userdata* de tipo incorreto. Para impedir o código Lua de alterar a função de coleta de um objeto *userdata*, pode-se atribuir um valor qualquer, como `false`, ao campo `__metatable` da metatabela, que passa a ser retornado no lugar da metatabela, tornando a metatabela em si inacessível.

Outro recurso relacionado à gerência de memória oferecido por Lua é a possibilidade de configurar, em tempo de execução, a função de alocação de memória a ser usada pela máquina virtual. Na criação de um novo estado Lua, passa-se como parâmetro uma função de alocação que deve oferecer funcionalidade similar às funções C `free` e `realloc`, dependendo se o tamanho de bloco passado for igual ou maior do que zero.

Lua oferece uma interface com o coletor de lixo através das funções `lua_gc` em C e `collectgarbage` em Lua. O coletor de Lua implementa *mark-and-sweep* incremental e permite ao programador configurar parâmetros relacionados aos intervalos de coleta, bem como ativar, desativar, disparar ciclos completos e executar passos do coletor.

3.2.5 Perl

Como Python, Perl realiza coleta de lixo baseada em contagem de referências. A API provê funções para o controle explícito da contagem: `SvREFCNT_inc` e `SvREFCNT_dec` para incremento e decremento e um *getter*, `SvREFCNT`. Outra forma de alterar a contagem de referências de um valor é atribuindo-o a uma referência Perl com `newRV_inc`. A contagem do valor referenciado será incrementada, fazendo com que – salvo tenha sua contagem alterada explicitamente – este se mantenha válido enquanto for referenciado pelo `RV`. É importante notar, porém, que as funções da API que criam valores, como `newSViv`, inicializam os seus contadores de referências com 1. Isto tem o efeito de que se um valor é criado em uma função C, armazenado em um `RV` com `newRV_inc` e esta referência é retornada a Perl, o valor nunca será coletado, pois o seu contador não retornará a 0 quando a referência for destruída. A forma correta, então, é usar `newRV_noinc` para `RV`s contendo valores recém-criados e `newRV_inc` quando um `RV` deve manter válido um valor já existente.

A inicialização da contagem de referências em 1 garante que valores criados continuarão válidos durante a execução de uma função C sem que seja necessário armazenar o valor no espaço de Perl. Estes valores podem também ser armazenados em variáveis globais e estruturas de dados de C e se manterão

válidos até que a sua contagem de referências seja decrementada. Para valores cujo tempo de vida é restrito a uma função, a API de Perl define o conceito de valores "mortais" como uma forma de permitir desalocar todos os valores temporários de uma função de uma só vez. Um `SV`, `AV` ou `HV` pode ser criado com `sv_newmortal` ou, mais comumente, convertido para mortal com `sv_2mortal`. Na prática, marcar um valor como mortal corresponde a indicar que ele deve ter a sua contagem de referências decrementada na chamada da macro `FREETMPS` ao término da função, conforme será visto na Seção 3.3.5. Algumas funções da API retornam valores mortais: por exemplo, `hv_delete` remove um elemento de uma hash e, caso a flag `G_DISCARD` não seja passada, retorna o elemento removido como um `SV` mortal.

A API de Perl não possui facilidades de interface com o coletor de lixo, mas possui algumas funções para auxílio em depuração que reportam informações sobre o estado da coleta de lixo. A função `sv_report_used` exibe o conteúdo de todos os `SV`s do interpretador. O módulo `Devel::Peek` permite examinar a partir de Perl o conteúdo de valores (contagem de referências, *flags*, etc.) – a partir de C, estas informações estão disponíveis diretamente já que as estruturas não são opacas.

3.2.6 Comparação

Coleta de lixo tem por objetivo isolar, na medida do possível, o programador da gerência de memória. Desta forma, idealmente uma API deveria também ser tão independente quanto possível do algoritmo de coleta de lixo utilizado pela implementação da máquina virtual. Perl e Python realizam coleta de lixo baseada em contagem de referências, e isto transparece nas operações de incremento e decremento de referências freqüentemente necessárias durante o uso de suas APIs.

Ruby utiliza um contador de referências baseado em mark-and-sweep. A sua API consegue abstrair bem este fato para a manipulação de objetos nativos Ruby, mas a implementação do coletor é evidente na criação de tipos Ruby em C, onde precisamos declarar uma função de marcação quando temos estruturas C que guardam referências para objetos Ruby. A API de Lua vai além ao isolar-se da implementação do coletor de lixo utilizado: o único ponto da API onde o emprego de um coletor de lixo incremental é aparente é na rotina de interação direta com o coletor, `lua_gc`, onde pode-se configurar parâmetros deste.

Das cinco linguagens, a única cuja API abstrai totalmente a implementação do coletor de lixo é Java. A única operação de interface com o coletor

que a linguagem provê, `System.gc()`, não recebe parâmetros e não especifica como ou quando a coleta deve ser feita[14]. De fato, as várias implementações da JVM utilizam algoritmos diferentes para coleta de lixo.

Na manipulação de dados através da API, Lua e Ruby são as linguagens que menos demandam do programador preocupações com gerência de referências. Ruby mantém o controle das referências retornadas a funções C varrendo a pilha de C durante a coleta de lixo, detectando a presença de referências armazenadas em variáveis locais. Lua evita o problema como um todo, mantendo seus objetos na pilha virtual e não retornando referências a eles ao código C.

O problema de referências armazenadas em variáveis locais de uma função é tratado por Perl e Java de forma semelhante, definindo dois tipos de referências, globais e locais (referências locais são chamadas de "variáveis mortais" em Perl). Referências locais têm gerência implícita (salvo alguns casos, como discutido na Seção 3.2.3). As funções da API de Java retornam referências locais por padrão, que podem ser convertidas para globais com `NewGlobalRef`. Em Perl ocorre o contrário, e as referências globais são convertidas para locais com `sv_2mortal`. O modelo de Java é mais interessante, pois normalmente são usadas mais variáveis de escopo local do que global. Valores armazenados globalmente têm sempre alguma forma de gerência explícita associada a si, mesmo em Ruby e Lua, através de `rb_global_variable` e `luaL_ref`/`luaL_unref`.

3.3 Chamada de funções a partir de C

A API deve prover uma forma de invocar a partir de C funções a serem executadas pela linguagem de script. Isto envolve a passagem de dados entre estes dois "espaços", conforme visto na Seção 3.1 e as implicações que isto traz sobre o tempo de vida dos objetos, abordadas na Seção 3.2. Devido à tipagem estática de C, não é possível usar uma sintaxe transparente para chamada de funções registradas em tempo de execução. É necessário então que a API defina funções para realizar chamadas na linguagem de script.

Nesta seção serão discutidas as facilidades oferecidas por cada API para a invocação de funções para execução na máquina virtual. As principais questões envolvidas são como referenciar a função a ser chamada, como passar argumentos a ela e como obter o valor de retorno, incluindo formas de notificação no caso de erros. Para fins de ilustração, será apresentado em cada linguagem um exemplo de chamada de uma função simples. Assume-se que

[14] A documentação é propositalmente vaga, dizendo apenas que este método *"suggests that the Java Virtual Machine expend effort toward recycling unused objects"* .

tenha sido definida no espaço de cada linguagem de script uma função `teste`, que recebe um inteiro e uma string como parâmetros e retorna um inteiro como resultado. Para maior brevidade, a verificação de erros será omitida nos exemplos.

3.3.1 Python

Para chamar uma função Python a partir de C, deve-se inicialmente obter um ponteiro para o `PyObject` correspondente à função, como visto na Seção 3.1.1. Além de funções implementadas em Python e funções C registradas a partir da API, qualquer tipo de dado que implemente o método `__call__` (ou declare uma função no campo `tp_call` de sua estrutura `PyTypeObject`) pode ser chamado como uma função.

A API de Python oferece diversas funções para realização de chamadas a partir de C. A função mais geral, `PyObject_Call`, recebe como parâmetros o objeto a ser chamado, uma tupla Python contendo os parâmetros a serem passados e opcionalmente um dicionário de argumentos *keyword*. Como conveniência, outras funções permitem passar os argumentos de outras formas. Por exemplo, `PyObject_CallFunction` encapsula a chamada a `Py_BuildValue` (vista na Seção 3.1.1), aceitando diretamente a string de formato desta e os valores a serem convertidos. `PyObject_CallFunctionObjArgs` é uma função *vararg* que aceita uma seqüência de ponteiros para `PyObject`s.

Existem também funções de conveniência para a invocação de métodos. A função `PyObject_CallMethod` é uma variante de `PyObject_CallFunction` que recebe como parâmetros um `PyObject` e uma string C contendo o nome do método. Assim, por exemplo, as duas formas abaixo são equivalentes à chamada Python `ret = uma_string.split(" ")`:

```
/* "s" indica que o parâmetro seguinte é do tipo string. */
PyObject* ret = PyObject_CallMethod(uma_string, "split", "s", " ");

PyObject* split = PyObject_GetAttrString(uma_string, "split");
PyObject* ret = PyObject_CallFunction(split, "s", " ");
```

É interessante notar que quando um método é chamado como função, o argumento `self` não é passado explicitamente.

O valor de retorno em todas as funções de chamada é um ponteiro para `PyObject`. Assim como ocorre em código Python, quando funções Python retornam múltiplos valores, estes são encapsulados em uma tupla. Para funções que não retornam valor, as funções C devem retornar `Py_None`. Em caso de erro

na chamada, as funções retornam `NULL`. A ocorrência de exceções pode então ser verificada com a função `PyErr_Occurred`.

Uma forma típica para chamar uma função Python `teste`, incluindo a obtenção da função e a conversão dos valores de entrada e saída entre Python e C, é exibida a seguir:

```
PyObject* globals = PyModule_GetDict(PyImport_AddModule("__main__"));
PyObject* teste = PyDict_GetItemString(globals, "teste");
/* "si" indica que seguem parâmetros string e inteiro */
PyObject* obj_result = PyObject_CallFunction(teste, "si", "entrada", 2);
/* Converte o valor para C */
long result = PyInt_AsLong(obj_result);
/* Libera o PyObject temporário retornado. */
Py_DECREF(obj_result);
```

Uma função global é obtida através do dicionário do módulo `__main__`. A conversão dos dados de entrada de C para Python é feita através da string de formato recebida por `PyObject_CallFunction`. Esta chamada equivale a `obj_result = teste("entrada", 2)` em Python. O valor de saída é retornado como uma nova referência a um objeto Python e, desta forma, precisa ter a contagem de referências decrementada após o seu uso. As funções `PyImport_AddModule`, `PyModule_GetDict` e `PyDict_GetItemString` retornam referências emprestadas, portanto a contagem de referências dos `PyObject`s retornados por elas não precisam ser decrementadas após o seu uso. Todavia, após a chamada da função Python, não há garantia de que os ponteiros `globals` e `teste` ainda apontem para objetos válidos – precisaríamos ter incrementado a sua contagem de referências caso quiséssemos usá-los novamente.

3.3.2 Ruby

Como métodos não são valores de primeira classe em Ruby, eles não são representados como `VALUE`s na sua API de C. Para a chamada de métodos Ruby em C, a API oferece a função `rb_funcall` e algumas variantes. Em comum, todas recebem como parâmetro o `VALUE` indicando ao objeto sobre o qual o método se refere, um `ID` referente à string internalizada contendo o nome do método e um inteiro informando o número de argumentos.

Como em Python, as funções da API para invocação de métodos diferem na forma como os argumentos são passados. Por exemplo, a função `rb_funcall` recebe os argumentos na forma de `VALUE`s passados como varargs C; `rb_funcall2` recebe um array C de `VALUE`s; `rb_apply` recebe um `VALUE` que deve ser um array Ruby contendo os parâmetros. Todas elas retornam

um `VALUE` como parâmetro. Assim como em código Ruby, múltiplos valores de retorno são representados como um array Ruby.

Todas as rotinas de chamada de função na API se referem a métodos, precisando assim especificar um objeto sobre o qual o método deve ser aplicado. Funções globais em Ruby são definidas como métodos do módulo `Kernel`, que é incluído pela classe `Object` e são, assim, acessíveis a partir de qualquer objeto, incluindo `nil`. Desta forma, pode-se invocar funções globais passando a constante C `Qnil` como o objeto alvo do método.

A seguir, é apresentada a forma típica para chamar em C uma função global Ruby `teste`, novamente incluindo a conversão dos valores de entrada e saída entre C e o interpretador.

```
ID teste = rb_intern("teste");
VALUE val_result = rb_funcall(Qnil, teste, 2,
                              rb_str_new2("entrada"), INT2NUM(2));
long result = NUM2LONG(val_result);
```

Diferentemente de Python, não é necessário obter uma referência para a função, bastando passar o nome desta na forma de `ID` e o objeto a que ela se refere (no caso, `Qnil`, indicando uma função global). A conversão dos dados de entrada de C para Ruby é feita através da função `rb_str_new2` e da macro `INT2NUM`, que criam `VALUE`s.

Conforme discutido na Seção 3.2.2, o controle da validade de `VALUE`s é feito implicitamente. Assim, pode-se fazer a chamada de funções que criam `VALUE`s diretamente na passagem de parâmetros de `rb_funcall`. De fato, as três linhas acima poderiam ter sido condensadas, passando a chamada `rb_funcall` como parâmetro para `NUM2LONG` e a chamada `rb_intern` como segundo parâmetro de `rb_funcall`, tendo sido separadas apenas para maior legibilidade.

Um tipo de dados que possui um tratamento um tanto irregular em Ruby é o de blocos de código. A sintaxe em Ruby para a declaração de blocos é especial: blocos só podem ser definidos como o último argumento de uma chamada de método. Assim, eles não são valores de primeira classe, não podendo ser, por exemplo, declarados em uma atribuição a variável. Eles podem, no entanto, ser promovidos a valores de primeira classe, na forma de objetos da classe `Proc`. Isto pode ser feito de duas formas: explicitamente, passando um bloco para o método `Proc.new`, ou implicitamente, quando o bloco for passado para um método que declara um último parâmetro formal precedido de `&`. Esta variável conterá o bloco convertido para um `Proc`. Na chamada de funções que esperam blocos, `&` converte o `Proc` para um bloco. Objetos `Proc` podem ser manipulados através da API de C como

qualquer outro objeto Ruby, mas não há correspondente na API de C para a funcionalidade do operador `&` em chamadas de função.

O status especial dos blocos de código complica a sua manipulação a partir de código C, e em particular a invocação de métodos que os esperam como parâmetro. Digamos que queremos invocar o seguinte método Ruby a partir de C:

```
def uma_funcao_ruby()
   print("uma_funcao_ruby vai invocar o bloco.\n")
   yield
   print("uma_funcao_ruby encerrou.\n")
   return 42
end
```

A função espera que um bloco de código seja passado para que seja invocado via o comando `yield`. Como vamos invocar a função a partir de C, queremos também passar código C como um bloco, representado na seguinte função:

```
VALUE um_bloco_C() {
   fprintf(stderr, "um_bloco_C está rodando.\n");
}
```

A conversão de objetos `Proc` para blocos proporcionada pelo operador `&` em Ruby não possui equivalente na API C. Assim, `rb_funcall` não é capaz de passar `Proc`s para funções que aceitam blocos. A forma intuitiva de fazer a chamada de funções Ruby a partir de C, neste caso, então, não funciona:

```
ID uma_funcao_ruby = rb_intern("uma_funcao_ruby");
/* O segundo parâmetro é um argumento adicional a ser
   opcionalmente passado na invocação de um Proc */
VALUE um_proc = rb_proc_new(um_bloco_C, Qnil);
/* Não funciona! Um Proc não é um bloco de código. */
VALUE resultado = rb_funcall(Qnil, uma_funcao_ruby, 1, um_proc);
```

As únicas formas de invocar um método Ruby passando um bloco de código são através de `rb_eval_string` e `rb_iterate`. A primeira abordagem, além do custo de desempenho causado pelo *parsing* da string de código, tem a inconveniência de exigir o uso de variáveis temporárias para que se possa obter os valores de retorno de volta ao espaço de C. No modelo usando `rb_eval_string`, a função C que atuará como bloco deve ser declarada no espaço de Ruby. Há duas alternativas de como fazer isto: registrando o método em Ruby e invocando-o em um bloco *wrapper* declarado na string de código Ruby:

```
/* Declara uma função global com 0 parâmetros de entrada */
rb_define_global_function("um_bloco_C", um_bloco_C, 0);
rb_eval_string("$resultado = uma_funcao_ruby { um_bloco_C() }");
VALUE resultado = rb_gv_get("$resultado");
```

Ou encapsulando a função em um objeto `Proc` a partir de C com `rb_proc_new` e então usando a notação `&` na string de código Ruby avaliada:

```
VALUE um_proc = rb_proc_new(um_bloco_C, Qnil);
rb_gv_set("$um_proc", um_proc);
rb_eval_string("$resultado = uma_funcao_ruby(&$um_proc)");
VALUE resultado = rb_gv_get("$resultado");
```

A segunda abordagem explora o fato de que a única função da API de C capaz de produzir blocos de código diretamente é `rb_iterate`. Esta função recebe dois ponteiros de função, um para a função a ser invocada e outro para a função que atuará como o bloco de código; chamadas a `yield` dentro da primeira função invocarão a segunda. O bloco pode quebrar o fluxo de execução com `rb_iter_break`. Passando como "função de iteração" para `rb_iterate` uma função *wrapper* que simplesmente chama o método Ruby desejado com `rb_funcall`, é possível simular uma chamada a `rb_funcall` que recebe uma função C como bloco de código.

```
VALUE chama_uma_funcao_ruby() {
   ID uma_funcao_ruby = rb_intern("uma_funcao_ruby");
   return rb_funcall(Qnil, uma_funcao_ruby, 0);
}
...
/* Os argumentos Qnil indicam que não há parâmetros
   nem para a função nem para o bloco */
VALUE resultado = rb_iterate(chama_uma_funcao_ruby, Qnil,
                             um_bloco_C, Qnil);
```

Note que nenhum argumento é passado em `rb_funcall` – a função `rb_iterate` define `um_bloco_C` como sendo o "bloco de código atual" e esta definição é herdada implicitamente por `rb_funcall`.

Para o caso comum de realizar iterações sobre o método `each` de coleções, Ruby oferece uma função *wrapper* `rb_each`. Esta função foi projetada para ser passada como primeiro argumento de `rb_iterate`. Funções C executando como bloco de código podem quebrar o fluxo de execução com `rb_iter_break`. O mecanismo de *yield*, tanto para código C quanto para chamadas nativas em Ruby, é implementado usando as funções C `setjmp` e `longjmp`.

Para o correto tratamento de erros, funções C que realizam chamadas a funções Ruby devem ser encapsuladas em uma chamada `rb_protect` ou em

uma de suas variantes, `rb_ensure` e `rb_rescue`. Esta função captura exceções disparadas por código Ruby (ou código C usando `rb_raise`). Caso o programa não use `rb_protect`, exceções em Ruby resultarão em erros fatais.

3.3.3 Java

De forma similar ao acesso a atributos, na chamada de métodos Java a partir de C deve-se inicialmente obter um identificador para o método, do tipo `jmethodID`. Estes identificadores são tipicamente obtidos com a função `GetMethodID`, que recebe como parâmetros a classe (instância de `jclass`) e duas strings, uma com o nome do método e outra com a assinatura do método. A sintaxe que descreve assinaturas de métodos é similar à de descritores de campos discutida na Seção 3.1.3. Parâmetros são listados entre parênteses, seguidos do tipo de retorno. Por exemplo, `"([Ljava/lang/String;II)V"` indica uma função com parâmetros `String[], int, int` e retorno `void`. Alternativamente ao uso de `GetMethodID`, a partir de Java 1.2 é possível obter um `jmethodID` correspondente a um método aplicando a função `FromReflectedMethod` sobre um objeto Java do tipo `Method` – isto é, um método reificado através da API de reflexão de Java.

Uma vez obtido o `jmethodID`, um método pode ser invocado através de alguma das 90 funções da família `Call*Method*`. Os nomes das funções seguem a forma

`Call`*[tipo]*`[retorno]``Method`*[argumentos]*

O *[tipo]* pode ser `Static` para funções estáticas, passando na chamada uma `jclass` como parâmetro; `Nonvirtual` para invocar implementações de um método em uma classe específica sobre um determinado objeto, passando uma `jclass` e um `jobject` como parâmetros; ou omitido para métodos de instância, passando o `jobject` sobre o qual o método será aplicado. O tipo de retorno é indicado em *[retorno]*: `Void`, `Object`, `Int`, etc.

Os argumentos do método podem ser passados de três formas: como *varargs*, como um array C de `jvalue`s, ou propagando uma `va_list` recebida. Por exemplo, na forma mais simples, um método de instância sem retorno e sem parâmetros é invocado com `CallVoidMethod`. Já `CallStaticIntMethodA` chama um método estático que retorna um `jint` com a lista de parâmetros passada em um array de `jvalue`s. Como Java é uma linguagem estaticamente tipada, não é preciso especificar o número ou o tipo dos parâmetros passados nas funções de chamada de métodos. Estas informações já estão especificadas nos `jmethodID`s.

É importante notar que, ao obter identificadores de métodos e campos fazendo a resolução a partir do `jobject` recebido na variável `this` e do nome do método ou campo, com `GetObjectClass` e `GetFieldID`, estamos efetivamente resolvendo nomes através de escopo dinâmico. Isto implica que, por exemplo, caso um método `Pai.metodo` implementado em C acesse um atributo privado `umAtributo` e uma subclasse `Filho` também defina um atributo privado `umAtributo`, a chamada a esse método em uma instância `f` de `Filho` acabaria por acessar `Filho.umAtributo` e não `Pai.umAtributo`. Este comportamento é diferente do que ocorreria se `Pai.metodo` fosse implementado em Java, onde o *binding* de membros privados é definido lexicamente. Para garantir à implementação em C de `Pai.metodo` que o atributo `umAtributo` acessado é realmente `Pai.umAtributo`, deve-se armazenar no espaço de C o identificador do campo obtido a partir da `jclass` de `Pai` – obtida, por exemplo, em uma função `static native`.

O código C pode verificar a ocorrência de exceções através de `ExceptionCheck` e optar por tratá-la, obtendo uma referência local da exceção com `ExceptionOccurred` e posteriormente zerando-a com `ExceptionClear`, ou então mantê-la ativa de modo que seja propagada ao código Java.

Para realizar o exemplo da função `teste`, já que Java não possui funções globais, vamos assumir que `teste` é um método estático de uma classe chamada `Exemplo` e que estamos rodando o código C a seguir em um contexto onde possuímos uma referência a um ambiente de execução Java chamado `J` (este ponteiro, do tipo `JNIEnv`, será discutido na Seção 3.4.3).

```
jclass exemplo = (*J)->FindClass(J, "Exemplo");
jmethodID teste = (*J)->GetStaticMethodID(J, exemplo,
                        "teste", "(Ljava/lang/String;I)I");
jstring entrada = (*J)->NewStringUTF(J, "entrada");
long retorno = (*J)->CallStaticIntMethod(J, exemplo, teste,
                                         entrada, (jint)2);
```

Inicialmente, é obtida uma referência à classe `Exemplo`, a partir da qual requisitamos o identificador do método desejado, baseado no seu nome e assinatura. Como em Ruby, a string passada como parâmetro deve ser convertida para um tipo da máquina virtual. Já para o segundo argumento e para o valor de retorno, exploramos o fato de que o tipo `jint`, correspondente ao tipo Java `int` (inteiro de 32 bits), é compatível com o tipo `long` de C (inteiro de pelo menos 32 bits). Todas estas funções da API retornam referências locais, que serão liberadas automaticamente ao final da função C onde as chamadas à API foram feitas.

3.3.4 Lua

Tanto em funções C disparadas por Lua como em chamadas de funções Lua realizadas a partir de código C, os parâmetros de entrada e os valores de retorno são passados através da pilha virtual apresentada na Seção 3.1.4.

Para chamar uma função Lua a partir de C, devemos inicalmente empilhar o objeto Lua referente a ela: para funções globais, obtendo-a com `lua_getglobal`, para funções armazenadas em tabelas, com `lua_gettable`. A seguir, empilhamos os seus parâmetros e então invocamos `lua_call` ou `lua_pcall`, indicando quantos valores da pilha devem ser passados como parâmetro. A diferença entre as duas funções está no tratamento de erros: `lua_call` propaga os erros sinalizados, usando `longjmp`; `lua_pcall` captura os erros, retornando um código de *status* e a mensagem de erro na pilha.

No caso de execução sem erros, a pilha conterá os valores de retorno da função chamada. O número de valores de retorno pode ser explicitamente requisitado na chamada de `lua_call` ou `lua_pcall`, ou ser definido em tempo de execução, requisitando o valor especial `LUA_MULTRET`. Se um número de valores de retorno for solicitado e este não for passado pela função chamada, o número de valores será ajustado adicionando elementos `nil` ou descartando valores em excesso. Para chamadas com `LUA_MULTRET`, todos os valores são empilhados. Nesse caso, a única forma de descobrir quantos valores foram retornados é comparando o tamanho da pilha antes e depois da chamada.

A função `lua_cpcall` permite chamar funções C realizando captura de erros de forma similar a `lua_pcall` sem precisar registrá-las como valores Lua. Esta funcionalidade é similar à oferecida por `rb_protect` em Ruby. Ruby, todavia, não oferece função análoga a `lua_pcall`, sendo às vezes necessário encapsular a chamada de funções Ruby em funções C que obedeçam à assinatura esperada por `rb_protect`.

Lua não possui distinção entre funções e métodos, mas possui açúcar sintático que permite invocar funções armazenadas em tabelas com uma sintaxe de chamada de métodos: `t:m(x)` significa `t.m(t,x)`. Todavia, não há na API de C uma chamada específica para replicar esta abreviação. Para funções armazenadas em tabelas, a função deve ser obtida com `lua_gettable` e a tabela deve ser empilhada explicitamente juntamente com os demais parâmetros.

O exemplo da chamada da função `teste` demonstra a disciplina de pilha adotada na API de Lua. Similarmente ao exemplo de Java na Seção 3.3.3, iremos assumir a existência de um ponteiro `L` do tipo `lua_State`, que será explicado mais adiante na Seção 3.4.4.

```
lua_getglobal(L, "teste");            /* Empilha a função teste */
lua_pushstring(L, "entrada");         /* Empilha a string "entrada" */
lua_pushinteger(L, 2);                /* Empilha o número 2 */
lua_call(L, 2, 1);                 /* Chama a função com 2 parâmetros,
                                          e espera 1 como retorno */
long retorno = lua_tointeger(L, -1); /* Obtém o resultado no
                                          topo da pilha (-1) */
lua_pop(L, 1);                        /* Remove-o da pilha */
```

Com `lua_getglobal`, é empilhada a função global `teste`. Em seguida, os dois argumentos de entrada são empilhados. A função é então invocada com `lua_call`, indicando dois parâmetros de entrada e um de saída. O valor de retorno, no topo da pilha (índice -1) é convertido para C com `lua_tointeger`. Esta última função não remove o valor da pilha: para retorná-la ao seu estado inicial, precisamos removê-lo explicitamente com `lua_pop`. Como em nenhum momento a API retorna ponteiros para objetos Lua, não há preocupações com coleta de lixo.

3.3.5 Perl

A chamada de funções Perl a partir de C se dá através de uma disciplina de pilha, como em Lua. Parâmetros de entrada são especificados através de operações de empilhamento e valores de retorno são obtidos na pilha após a chamada da função. As funções `call_sv`, `call_pv` e `call_method` variam apenas na forma como a função a ser chamada é especificada: através de um `SV`, de uma string C ou de uma string C descrevendo o nome de um método de algum objeto ou classe inserido na pilha. A função `call_argv`, como conveniência, recebe como um argumento adicional um array C contendo strings C representando parâmetros a serem empilhados. Todas retornam o número de valores de retorno disponíveis na pilha.

As funções `call_*` possuem um argumento de flags a serem passadas que indicam a forma que a função deve ser chamada e como tratar os parâmetros de entrada e valores de retorno. `G_VOID`, `G_SCALAR` e `G_ARRAY` especificam o contexto como a função deve ser chamada. Em contextos escalares, por exemplo, somente um escalar é retornado na pilha; se a função chamada retornar uma lista, somente o último elemento desta estará disponível na pilha. `G_DISCARD` indica que os valores de retorno devem ser automaticamente descartados; `G_NOARGS` indica que o array padrão de parâmetros `@_` não deve ser construído[15].

[15]Isto tem o efeito colateral de que a função chamada herda o valor de `@_` da função chamadora.

O precedimento para verificação de erros depende do contexto e das flags passadas, que afetam como situações de erro são reportadas no valor de retorno das funções `call_*` e nos valores retornados na pilha. A flag `G_EVAL` encapsula a chamada em um bloco `eval`, capturando erros. Assim, a ocorrência de erros pode ser verificada através da macro `ERRSV`, que retorna o `SV` contendo a mensagem de erro. Acrescentando a flag `G_KEEPERR`, mensagens de erro não sobrescrevem a variável especial `$@`, mas são concatenadas a ela, acumulando seqüências de erros em níveis de chamada diferentes.

Uma série de macros descrevem um protocolo para a chamada de funções e a manipulação de parâmetros de entrada e saída. As principais serão explicadas a seguir, na apresentação da versão Perl da chamada da função `teste`:

```
dSP;
ENTER;
SAVETMPS;
PUSHMARK(SP);
XPUSHs(sv_2mortal(newSVpv("entrada", 0)));
XPUSHs(sv_2mortal(newSViv(2)));
PUTBACK;
call_pv("teste", G_SCALAR);
SPAGAIN;
long result = POPl;
PUTBACK;
FREETMPS;
LEAVE;
```

Inicialmente `dSP` declara uma cópia local do ponteiro da pilha de Perl. Em seguida, `ENTER` e `SAVETMPS` criam um escopo para valores mortais. `PUSHMARK` inicia a contagem de parâmetros a serem passados para a função. Estes parâmetros são então empilhados com `XPUSHs`. Os valores criados com `newSVpv` e `newSViv` são convertidos para valores mortais com `sv_2mortal`, para que não tenham que ter sua contagem de referências decrementada explicitamente após a chamada da função. `PUTBACK` encerra a contagem de parâmetros. É feita então a chamada da função global Perl `teste`, em contexto escalar, com `call_pv`.

Após o retorno desta função, a memória da pilha de Perl pode ter sido realocada, mudando o endereço do ponteiro de pilha obtido inicialmente com `dSP`. Para certificar-se que o seu valor está correto, deve-se chamar `SPAGAIN` após funções `call_*`. A função `POPl` desempilha um valor e o converte para `long` (há funções similares para outros tipos, como `POPs` para `SV`s e `POPpx` para strings). Estas operações desempilham valores atualizando a cópia local do ponteiro de pilha. Assim, `PUTBACK` deve ser chamado novamente para atualizar

o ponteiro global. Finalmente, `FREETMPS` e `LEAVE` decrementam a contagem de referências dos valores mortais.

3.3.6 Comparação

Em Python, Lua e Perl, funções podem ser acessadas como objetos da linguagem e invocadas. Em Ruby e Java, a API define tipos especiais usados para referenciar métodos. Como na manipulação de dados, Python oferece uma API extensa, com diversas funções de conveniência permitindo passar argumentos como tuplas Python, objetos Python passados como *varargs*, valores C a serem convertidos pela função de chamada, etc. Java também oferece um grande número de funções para invocação de métodos e, devido à tipagem estática da linguagem, os parâmetros de entrada podem ser passados como *varargs* de forma direta, sem precisar especificar a forma como a conversão deles deve ser realizada. Ruby também oferece algumas variantes de funções de chamada.

Lua, em contraste, separa a rotina da chamada da função da passagem dos parâmetros, que é feita anteriormente através da pilha. Isto é uma solução bastante simples, mas o código resultante é menos claro que as chamadas equivalentes em linguagens como Ruby e Python. Perl também faz chamadas de função utilizando um modelo de pilha, mas ao contrário de Lua o seu uso é demasiadamente complexo, por exigir um protocolo de macros que expõem o funcionamento interno do interpretador. Outro complicador é o tratamento de valores de retorno, pois estes variam de comportamento conforme o contexto Perl em que a função é chamada.

Em Lua e Python, a ocorrência de erros pode ser verificada com o valor de retorno da função. De modo parecido, Perl permite detectar erros na chamada mais recente verificando uma variável especial; em Java, isto é feito chamando uma função da API. Somente o tratamento de erros em Ruby é mais convoluto, pois estranhamente a API oferece uma função que permite invocar funções C em modo protegido, mas não uma equivalente que permita chamar funções Ruby. Torna-se necessário escrever uma função *wrapper* nestes casos, o que será ilustrado na Seção 4.2.5.

3.4 Registro de funções C

Para permitir a invocação de funções C a partir de código da linguagem de script, a API deve fornecer uma forma de registrar estas funções no ambiente de execução. Em linguagens de tipagem estática, como Java, para que seja possível

chamar funções externas usando uma sintaxe igual à de funções nativas, o conjunto de funções externas deve ser declarado *a priori* de alguma forma. Já em linguagens com tipagem dinâmica, como é o caso de Python, Lua, Ruby e Perl, as funções podem ser usadas diretamente, bastando que sejam definidas em algum momento da execução antes de sua chamada. Assim, pode-se declarar as funções externas em tempo de execução através de código C usando a API da linguagem de script.

Também nesta seção, a apresentação de cada linguagem encerrará com um exemplo. Uma função C que, assim como nos exemplos da seção anterior, recebe um inteiro e uma string e retorna um inteiro, será registrada. Apresentaremos também, para cada linguagem, a forma de registrar a função como a global[16] `teste` de modo que ela possa ser invocada diretamente a partir da linguagem ou através da API.

3.4.1 Python

Python não possui um tipo "função" propriamente dito declarável em C. Métodos de classes, no entanto, são objetos e possuem um tipo específico, que pode ser verificado com a função `PyMethod_Check`. Tipicamente, métodos são criados passando-se um array de estruturas `PyMethodDef`. Estas estruturas são compostas do nome da função, o ponteiro da função C, um vetor de flags e uma string de documentação. As flags são usadas para indicar a convenção adotada para os parâmetros de entrada na função C. As flags mais comuns são: `METH_NOARGS`, usada para funções Python que não recebem parâmetros, indicando que a função C deve receber um único ponteiro para `PyObject`, que irá conter o `self` do método; `METH_VARARGS`, para funções que recebem como segundo parâmetro uma tupla Python que conterá um número variável de parâmetros passados de Python para C; e `METH_KEYWORDS`, para indicar que a função C recebe ainda como terceiro parâmetro um dicionário contendo os argumentos *keywords* passados à função.

De posse destas informações, funções da API que operam sobre arrays de `PyMethodDef` podem criar e associar objetos do tipo método no espaço de Python. `Py_InitModule`, por exemplo, inicializa um módulo com funções de um array de `PyMethodDef`. Similarmente, os métodos de uma classe implementada em C podem ser dados no campo `tp_methods` da estrutura `PyTypeObject` relativa à classe.

Funções C registradas em Python devem retornar um ponteiro para `PyObject`, ou `NULL` em caso de erro (opcionalmente declarando uma exceção

[16] Ou no caso de Java, método estático.

com `PyErr_SetString` ou `PyErr_SetObject`). Funções que não retornam valores devem retornar o objeto pré-definido `Py_None`, lembrando das questões de contagem de referência de valores retornados discutidas na Seção 3.2.1.

Embora métodos sejam usualmente criados em C usando estruturas `PyMethodDef`, é possível ainda criar um objeto do tipo método explicitamente a partir de C com a função `PyMethod_New`, passando como parâmetro um objeto Python "chamável" e o objeto ou classe a que ele deve se referir. Como visto na Seção 3.3.1, objetos Python podem ser tornados "chamáveis" implementando um método `__call__` em Python ou associando uma função C ao campo `tp_call` do seu `PyTypeObject` correspondente.

Uma implementação simples de uma função C que pode ser registrada em Python como a função global `teste` é dada a seguir:

```
PyObject* teste_py(PyObject* self, PyObject* args) {
   char* entrada; long n;
   /* Em caso de erro nos argumentos, PyArg_ParseTuple
      gera uma exceção apropriada automaticamente */
   if (!PyArg_ParseTuple(args, "sl", &entrada, &n))
      return NULL;
   printf("Recebi: %s e %ld \n", entrada, n);
   return PyInt_FromLong(42);
}
```

Como os argumentos são recebidos em uma tupla no segundo parâmetro, a assinatura de função empregada corresponde à flag `METH_VARARGS`. Os parâmetros de entrada são convertidos para C e verificados com `PyArg_ParseTuple`. O valor de retorno é convertido do tipo nativo C para um `PyObject` com `PyInt_FromLong`, gerando uma nova referência.

A API de Python é projetada primariamente para o desenvolvimento de módulos de extensão para a linguagem. Embora existam diversas funções para registrar métodos em classes e inicializar módulos com listas de funções, não há uma forma direta para registrar funções globais na máquina virtual. Uma maneira possível é usando a rotina utilitária para lookup de métodos `Py_FindMethod` e inserindo o método retornado no dicionário do módulo global `__main__`:

```
PyObject* globals = PyModule_GetDict(PyImport_AddModule("__main__"));
static PyMethodDef teste_def[] = {
   { "teste", (PyCFunction) teste_py, METH_VARARGS, "um teste" },
   { NULL }
};
PyObject* teste_obj = Py_FindMethod(teste_def, NULL, "teste");
PyDict_SetItemString(globals, "teste", teste_obj);
```

Note que foi passado NULL para Py_FindMethod, indicando que não há um objeto do qual o método faz parte. O argumento self recebido pela função C teste_py será também NULL e pode ser ignorado. O array teste_def foi declarado static para garantir que o PyMethodDef continuará válido enquanto a função global estiver registrada, pois na criação de teste_obj um ponteiro para ele é armazenado internamente no objeto criado.

3.4.2 Ruby

Para que funções C possam ser chamadas a partir de Ruby, elas devem ser declaradas como métodos de alguma classe ou módulo, ou ainda como uma função global. Para isto, passa-se um ponteiro de uma função C e o número de argumentos que a função espera para uma das funções apropriadas da API de Ruby: rb_define_method, rb_define_module_function, rb_define_global_function ou rb_define_singleton_method. O número de parâmetros passado indica a assinatura esperada para a função C. Ruby suporta explicitamente funções C com até 15 argumentos; como alternativa, os valores especiais -1 e -2 indicam, respectivamente, que a função C irá receber os parâmetros na forma de um array C de VALUEs ou na forma de um VALUE correspondente a um array Ruby.

De forma que lembra a função PyArg_ParseTuple discutida na Seção 3.1.1, Ruby possui uma função projetada para simplificar o processamento dos valores de entrada em funções C: rb_scan_args. Esta função pode ser usada quando os parâmetros de entrada são recebidos em um array Ruby. Como PyArg_ParseTuple, ela é uma função vararg que recebe uma string de formato indicando o número de parâmetros a serem coletados. Por outro lado, ela não realiza verificação de tipo dos argumentos. A string de formato permite indicar o número mínimo e máximo de parâmetros aceitos e se os parâmetros excedentes devem ser coletados em um array Ruby.

Uma vez declarada no espaço de objetos de Ruby, uma função C pode ser chamada como qualquer outro método. A função C pode verificar se o código Ruby lhe passou um bloco de código através da função rb_block_given_p. O bloco pode então ser invocado com rb_yield, que recebe um VALUE como argumento. Para passar múltiplos argumentos para rb_yield, deve-se passar um array Ruby. Para obter um VALUE do tipo Proc produzido a partir do bloco de código recebido é preciso usar rb_scan_args, que possui funcionalidade similar à do operador & em declarações de funções Ruby.

Funções C implementado métodos Ruby devem sempre retornar um VALUE (Qnil quando não há retorno). Funções que retornam múltiplos valores

devem fazê-lo através de um array Ruby.

Dando continuidade à série de exemplos, a função global Ruby `teste` pode ser implementada da seguinte forma em C:

```
VALUE teste_rb(VALUE self, VALUE val_entrada, VALUE val_n) {
   char* entrada = StringValuePtr(val_entrada);
   long n = NUM2INT(val_n);
   printf("Recebi: %s e %ld \n", entrada, n);
   return INT2NUM(42);
}
```

A conversão dos `VALUE`s de entrada é feita com as macros `StringValuePtr` e `NUM2INT`. Não há código explícito para tratamento de erros na conversão pois estas macros disparam exceções que saem da função via `longjmp` caso a conversão não seja possível. Para a saída, é produzido um `VALUE` com a macro `INT2NUM`. O primeiro parâmetro de entrada é necessário segundo a convenção de assinaturas de função da API, mas para funções globais ele deve ser ignorado.

Como Ruby oferece uma função na API para a definição de funções globais, o registro de `teste` é bastante simples:

```
rb_define_global_function("teste", teste_rb, 2);
```

É indicado o nome da função no espaço de Ruby, a função C correspondente e o número de parâmetros que ela espera (não incluindo o parâmetro `self`).

3.4.3 Java

Métodos declarados em Java que não são implementados na própria linguagem devem ser declarados através de um protótipo incluindo o modificador `native`. Assim, `native` não se refere a uma implementação nativa em Java, mas ao código do método ter sido compilado com código nativo do ambiente de execução, em oposição a *bytecodes* da máquina virtual. A implementação do método, usualmente encapsulada em uma biblioteca dinâmica C, deve ser carregada antes de sua execução usando a chamada `System.loadLibrary` em Java, usualmente em um bloco `static` da classe correspondente. Para cada método `native`, uma função C correspondente deve ser definida na biblioteca carregada.

O utilitário `javah` gera arquivos de cabeçalho C a partir de classes Java, com os protótipos de funções C no formato especificado pela JNI. Este formato especifica não apenas a assinatura dos parâmetros de entrada e tipo de retorno, mas também o nome da função, para que o *loader* realize a ligação

entre a função C e o método Java na máquina virtual. As funções devem se chamar `Java_`*[nome da classe]* `_`*[nome do método]*. No caso de sobrecarga de funções, um sufixo é adicionado indicando o tipo dos parâmetros de entrada (por exemplo, `Java_Classe_metodo__DI` para a versão de `Classe.metodo` que aceita um `double` e um `int` como parâmetros).

Os parâmetros de entrada da função são um ponteiro para `JNIEnv`, que representa uma thread da JVM, um `jobject` representando o objeto sobre o qual o método é aplicado (ou uma `jclass` para métodos estáticos) e os demais parâmetros do método Java em suas representações C, discutidas na Seção 3.1.3. Como o tipos dos parâmetros passados são definidos estaticamente tanto em Java como em C, não é preciso realizar a verificação dos tipos dos dados recebidos no código C. A assinatura das funções que implementam métodos, especificados nos arquivos de cabeçalho gerados pela ferramenta `javah`, já declaram os tipos corretos.

O tipo de retorno corresponde ao tipo C equivalente ao tipo de retorno do método Java. Valores representados com tipos de referência podem ser retornados tanto com referências locais como globais. Além de tratar ou propagar erros como discutido na Seção 3.3.3, funções C podem também gerar exceções com `Throw` e `ThrowNew` e retornar imediatamente. O valor de retorno será ignorado quando a exceção for capturada no código Java.

Como o modo de expor à máquina virtual de Java funções implementadas em C é diferente do empregado nas linguagens vistas anteriormente, iniciaremos pela forma de declarar a função para o espaço de Java, para só então mostrar a implementação de `teste` em C. Na classe em Java, declaramos um método `native`:

```
public class Exemplo {
   static native int teste(String entrada, int n);
   // ...demais membros da classe
   static {
      System.loadLibrary("Exemplo");
   }
}
```

Após compilar esta classe podemos passá-la para o comando `javah`, que gerará um arquivo de cabeçalho C. Este arquivo conterá o nome e assinatura da função C que a JVM irá procurar na biblioteca que será carregada por `System.loadLibrary`[17]. Esta biblioteca deve implementar funções relativas aos métodos declarados como `native`.

[17] O parâmetro passado no código Java é usado como base na construção de um nome dependente de plataforma. Em sistemas Unix, por exemplo, o comando `System.loadLibrary("Exemplo")` carrega o arquivo `libExemplo.so`.

Abaixo, é dada uma implementação em C, usando o cabeçalho gerado por `javah`, para o método `teste`:

```
#include <jni.h>
#include <stdio.h>
/* Cabeçalho gerado por javah */
#include "Exemplo.h"

JNIEXPORT jint JNICALL
Java_Exemplo_teste(JNIEnv* J, jclass c, jstring obj_entrada, jint n) {
   const char* entrada = (*J)->GetStringUTFChars(J, obj_entrada, NULL);
   printf("Recebi %s e %ld \n", entrada, n);
   (*J)->ReleaseStringUTFChars(J, obj_entrada, entrada);
   return 42;
}
```

`JNIEXPORT` e `JNICALL` são macros definidas em `jni.h` para dar maior portabilidade ao código C resultante. Como o método foi declarado como `static` em Java, uma referência para a classe é recebida como parâmetro para a função. Os demais parâmetros correspondem aos parâmetros do método Java, e são dados nos tipos equivalentes da JNI. Conforme discutido na Seção 3.1.3, a JNI dá tratamentos diferentes para tipos de referência e tipos imediatos. Desta forma, somente o parâmetro `obj_entrada` precisa ser convertido para C; tanto `n` como o valor de retorno podem ser usados diretamente como tipos básicos de C.

A string obtida com `GetStringUTFChars` é convertida para UTF-8 a partir da representação Unicode interna de Java. O mesmo ponteiro pode ser retornado pela JVM a diferentes threads que solicitem a mesma string. Deste modo, o código C deve notificar a sua liberação explicitamente com `ReleaseStringUTFChars`.

3.4.4 Lua

Funções C expostas ao espaço de Lua devem ter o tipo `lua_CFunction`, recebendo como único parâmetro um ponteiro para uma variável do tipo `lua_State` e retornando um `int`. Um `lua_State` encapsula todo o estado da máquina virtual de Lua; múltiplos estados Lua podem ser mantidos em paralelo. Todas as funções da API *core* de Lua recebem um `lua_State` como primeiro parâmetro, exceto `lua_newstate`, que cria um `lua_State` novo.

Ao início da função C, os argumentos passados a ela encontram-se inseridos na pilha virtual. Como em funções Lua, não há verificação sobre o número de argumentos passados para uma função C invocada a partir de

Lua ou através da API. O código C pode verificar o número de argumentos passados inspecionando o tamanho da pilha recebida.

A biblioteca auxiliar provê ainda funções para verificar de forma mais conveniente o tipo dos argumentos passados. Funções da família `luaL_check*` (`luaL_checkint`, `luaL_checkstring`, etc.) verificam o tipo de elementos da pilha e os retornam, sinalizando erro caso o tipo do elemento não seja o requisitado. As funções `luaL_opt*` se comportam de forma similar, permitindo ainda indicar um valor padrão caso o elemento esteja ausente ou seja `nil`.

Valores de retorno também são passados pela função C de volta para Lua através da pilha virtual. O valor inteiro retornado da função C indica quantos elementos da pilha devem ser retornados à função chamadora. Os demais valores da pilha são descartados.

Uma função C do tipo `lua_CFunction` pode ser passada para o espaço de Lua através da função `lua_pushcfunction`. Lua possui também algumas funções de conveniência para registrar um conjunto de funções C de uma só vez. Do mesmo modo que ocorre no uso de arrays `PyMethodDef` em Python, a função `luaL_register` registra uma lista de funções, recebendo um array de estruturas `luaL_Reg` contendo nomes e ponteiros de função.

A função C implementando a função de exemplo `teste` é dada a seguir:

```
int teste_lua(lua_State* L) {
   /* Obtém o primeiro parâmetro */
   const char* entrada = luaL_checkstring(L, 1);
   /* Obtém o segundo parâmetro */
   long n = luaL_checkinteger(L, 2);
   printf("Recebi %s e %ld \n", entrada, n);
   /* Empilha o valor de retorno */
   lua_pushinteger(L, 42);
   /* Retorna um valor, do topo da pilha */
   return 1;
}
```

A função tem assinatura idêntica à definição de `lua_CFunction`. Os parâmetros de entrada são obtidos das posições 1 e 2 da pilha e os seus tipos são verificados usando as funções da biblioteca auxiliar `luaL_checkstring` e `luaL_checkinteger`. Estas funções sinalizam erro em caso de falha na conversão, causando um `longjmp` como em Ruby.

O tipo da string obtida é `const char*`, pois ela aponta um bloco de memória gerenciado pela máquina virtual. Em Lua, no entanto, não é preciso notificar a liberação da string explicitamente, pois esta permanece válida enquanto o valor estiver na pilha. Como as funções `luaL_check*` não

desempilham os parâmetros e a pilha é esvaziada implicitamente ao final da função C, a string C obtida permanecerá válida ao longo da função.

Ao fim da função `teste_lua`, o valor de retorno para Lua é empilhado usando `lua_pushinteger`. O valor de retorno da função em C, `1`, indica à máquina virtual que há um único valor de saída a ser obtido da pilha e usado como retorno da função em Lua.

A função é registrada em Lua criando um objeto Lua do tipo `function` a partir da função C e armazenando este objeto em uma variável global. Isto poderia ser feito com `lua_pushcclosure` e `lua_setglobal`, mas o arquivo de cabeçalho `lua.h` possui uma macro que encapsula estas duas chamadas. Assim, a função pode ser registrada simplesmente com:

```
lua_register(L, "teste", teste_lua);
```

Usando a função `lua_pushcclosure`, é possível associar a uma função C valores Lua que serão acessíveis à função sempre que esta for chamada, de forma similar a variáveis locais `static` C. Este recurso provê uma funcionalidade similar às *closures* de Lua, porém mais limitado: os valores associados são privados às funções C, enquanto em Lua duas *closures* definidas num mesmo escopo terão acesso às mesmas variáveis, isto é, alterações nos valores em uma afetarão a outra. Esta forma restrita, porém, já permite a implementação em C de "funções com estado", como iteradores e geradores. Uma vez registrada no espaço de Lua, funções C passam a ser vistos como valores do tipo `function`, de forma igual a funções Lua. De fato, `lua_pushcfunction` é um caso particular de `lua_pushcclosure` onde nenhum valor Lua é associado à função.

3.4.5 Perl

Como discutido na Seção 3.1.5, a interface entre Perl e C foi projetada tendo em mente que a ligação entre as funções C e o interpretador Perl é feita através de código gerado a partir de uma descrição feita em uma linguagem de mais alto nível, XS. Código XS consiste de declarações de assinaturas de função com uma sintaxe especial, indicando regras para conversão dos parâmetros de entrada e saída, e código C descrevendo a implementação das funções. XS é projetada para o desenvolvimento de extensões Perl incluindo funções implementadas em C: o resultado final da compilação do código gerado pelas ferramentas XS (`h2xs`, `xsubpp`) são código C e Perl que juntos descrevem um pacote Perl (um conjunto de variáveis e funções armazenados sob um *namespace* comum).

Existe uma API pública para manipulação de dados Perl no código C, mas esta consiste basicamente das estruturas internas da implementação do

interpretador expostas para uso pelo pré-processador XS, acrescida de macros para maior conveniência do programador. De fato, Perl não expõe uma API documentada para o registro de funções (Okamoto 2006a). Assim, não é prático para uma aplicação embutir um interpretador Perl e expor a ele um conjunto de funções C usando apenas código C. A saída é criar uma extensão Perl usando XS que expõe funções da aplicação e importar o pacote resultante no interpretador embutido. O emprego desta abordagem foi observada nos plugins para scripting Perl de diversas aplicações [18].

O utilitário `h2xs` gera um diretório contendo o esqueleto de um módulo Perl: um script gerador de Makefiles, arquivos `.xs` e `.pm` a serem acrescidos de código XS e Perl, além de arquivos auxiliares. Retomando o exemplo da função teste, ela seria declarada da seguinte forma em XS:

```
long teste(entrada, n)
        char* entrada
        int n
        CODE:
                printf("Recebi %s e %ld \n", entrada, n);
                RETVAL = 42;
        OUTPUT:
                RETVAL
```

O arquivo `.xs` é convertido para `.c` com `xsubpp`. O código C para a conversão dos parâmetros de entrada e saída é gerado automaticamente. Em alguns casos, entretanto, precisamos manipular os valores da pilha de Perl explicitamente, como descrito na Seção 3.1.5. Em funções vararg, por exemplo, os argumentos adicionais devem ser acessados diretamente na pilha. O código para registrar as funções do módulo também é gerado automaticamente.

XS cria uma variável `RETVAL` automaticamente para armazenamento do valor de retorno em código C. O conteúdo desta variável é convertido para um valor Perl pelo código C gerado. Para que funções que retornam arrays possam operar corretamente em contextos escalares, deve-se verificar o contexto em que a função é chamada com `GIMME_V` e então retornar um `SV` ou `AV` conforme o caso. A função, nestes casos, deve ser declarada com tipo de retorno `SV*`, fazendo com que os valores de C devam ser convertidos para `SV`s Perl explicitamente.

[18] Vim (`http://www.vim.org`), Gimp (`http://search.cpan.org/search?mode=dist&query=gimp`) e Gaim (`http://gaim.sourceforge.net`) são algumas aplicações que implementam plugins Perl através de extensões XS. No plugin de Xchat (`http://www.xchat.org`), não há arquivos `.xs`, mas os fontes `.c` incluem funções declaradas com a API não documentada e o código Perl equivalente ao arquivo `.pm` gerado por `xsubpp` declarado como uma string C avaliada com `eval_pv`, dando a entender que o plugin foi implementado como uma extensão usando XS e depois convertido para um único arquivo-fonte C.

A documentação alerta que, para o caso de `AV`s, deve-se declarar o valor de retorno como variável mortal[19].

Uma vez compilada a extensão usando os Makefiles gerados por `h2xs`, esta pode ser carregada e usada a partir de Perl:

```
use Teste;
$ret = Teste::teste("entrada", 2);
print $ret . "\n";
```

Para expor funções de uma aplicação C a um interpretador Perl embutido nela, devemos criar uma extensão que encapsula estas funções usando XS, ligar a extensão à aplicação e carregá-la. A carga é feita passando para o interpretador durante a sua inicialização uma função C contendo chamadas `newXS`. O módulo Perl `ExtUtils::Embed` possui uma rotina `xsinit` para gerar o código C desta função. Na prática, gerar o código da função com `xsinit` é a melhor abordagem, já que a inicialização depende de rotinas não documentadas (o exemplo de função de inicialização incluído na documentação de Perl (MacEachern 2006) está desatualizado).

3.4.6 Comparação

Python e Ruby oferecem ao programador várias opções de assinaturas de função C reconhecidas pela API, o que é prático, uma vez que pode-se escolher assim diferentes representações em C para os parâmetros de entrada (agregados em um array, recebidos um a um, etc.) conforme o seu uso na função. Lua oferece apenas uma assinatura de função possível para funções C a serem registradas na máquina virtual, mas isto é apropriado para o modelo de pilha usado por sua API.

Em Java, as assinaturas de função são criadas através da ferramenta `javah` – devido à tipagem estática, os tipos dos parâmetros de entrada passados por Java são convertidos automaticamente pela JNI, o que é bastante conveniente uma vez que evita operações explícitas de conversão e verificação de tipos na função. Por terem tipagem dinâmica, as demais linguagens oferecem funções na API específicas para realizar estas verificações. As assinaturas de função em Perl são criadas apenas através da ferramenta XS, mas diferentemente de Java elas não são expostas ao programador. Isto traz a inconveniência de termos que pré-processar o código C como uma extensão XS mesmo quando estamos embutindo Perl em uma aplicação.

[19]Este comportamento é descrito na documentação como *"an unfixable bug (fixing it would break lots of existing CPAN modules)"* (Roehrich 2006).

O registro de funções em Ruby e Lua é simples. Em Lua, em particular, trata-se de uma atribuição igual à de qualquer outro objeto. Já em Python, existem recursos para o registro de funções em lote, usando arrays `PyMethodDef` (Lua oferece recurso similar com `luaL_register`), mas não há uma forma simples de registrar uma única função. Tanto em Java como em Perl o registro de funções é feito de forma implícita, e em nenhuma das duas APIs há funções para registrar novas funções C durante a execução do programa.