4
Estudo de caso: LibScript

Nos capítulos anteriores, foram discutidas as principais questões envolvendo interfaces de linguagens para C e a forma como estas questões são tratadas pelas linguagens abordadas neste estudo. Neste capítulo, realizaremos uma comparação entre as APIs através de um exemplo concreto, de modo a colocar implementações em cada uma das linguagens lado a lado. O exemplo consiste em uma biblioteca genérica para scripting, chamada LibScript, e uma série de plugins que realizam a interface com as diferentes linguagens de script.

4.1
LibScript

LibScript é uma biblioteca projetada para tornar aplicações extensíveis através de scripting de uma forma independente de linguagem. Ela é baseada em uma arquitetura de *plugins*, de modo a desacoplar a aplicação dos ambientes de execução providos pelas diversas linguagens. A biblioteca principal provê uma API para scripting independente de linguagem, permitindo a uma aplicação registrar as suas funções e disparar scripts que utilizem estas funções. Esta biblioteca então invoca o plugin da linguagem apropriada para rodar o script (por exemplo, LibScript-Python para código Python). Desta forma, o desenvolvedor da aplicação permite ao seu usuário utilizar diferentes linguagens para scripting sem adicionar todas elas como dependências do programa.

A biblioteca principal disponibiliza recursos para o registro de funções C por parte da aplicação e para a chamada destas funções por parte dos plugins (permitindo aos scripts acessar as funções), além de funções para a transferência de dados entre as duas partes. É possível também invocar funções implementadas nas máquinas virtuais embutidas nos plugins, possibilitando assim que scripts escritos em diferentes linguagens possam interagir.

4.1.1
Arquitetura de LibScript

LibScript é composta de uma biblioteca dinâmica principal, `libscript`, e *plugins* para diferentes linguagens (Figura 4.1). A biblioteca principal é

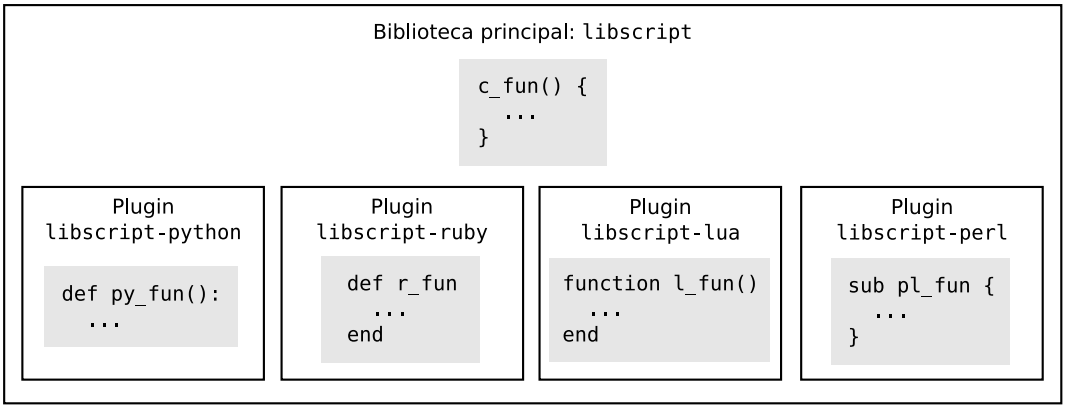

Figura 4.1: Visão geral da arquitetura de LibScript

ligada a uma aplicação, e expõe a ela uma API de scripting independente de linguagem, que permite executar arquivos, strings de código e invocar funções. Esta biblioteca é uma fina camada que encaminha estas operações para os plugins, que são bibliotecas dinâmicas auxiliares, carregadas em tempo de execução pela biblioteca principal. Estes plugins embutem os ambientes de execução das linguagens de script.

A aplicação pode registrar funções C na biblioteca principal (ilustrado pela função `c_fun` na figura) e solicitar a ela que execute scripts que registram funções nas diferentes linguagens. Todavia, a aplicação não interage diretamente com os plugins. Quando a biblioteca principal recebe código a ser executado em uma determinada linguagem, ela carrega o plugin adeqüado (caso este ainda não esteja carregado) e encaminha o código. O plugin irá executar o script em sua máquina virtual, o que pode registrar nela novas funções (ilustrado pelas funções `py_fun`, `r_fun`, `l_fun` e `pl_fun` na figura).

A biblioteca principal decide qual plugin carregar através de um identificador que especifica qual a linguagem do código a ser executado. Este identificador pode ser obtido a partir da extensão de arquivo de um script carregado, da linha de identificação "`#!`" no início do script[1] ou mesmo passado explicitamente pela aplicação.

Funções são registradas em LibScript em um *ambiente virtual*. A aplicação pode criar um ou mais ambientes na biblioteca principal, identificando-os com um nome. Um ambiente virtual ganha em cada plugin uma estrutura de dados específica da linguagem (classe, módulo, etc.) que o representará. No exemplo da Figura 4.2 temos dois ambientes virtuais criados pela aplicação na biblioteca principal, `X` e `Y`. Em cada um destes

[1]A linha "`#!`" é usada apenas para detectar a linguagem em que o script é escrito. Por exemplo, uma linha `#!/usr/bin/perl -w` indicará a carga do plugin `libscript-perl`, mas o interpretador Perl em `/usr/bin` não é usado e nem a flag `-w` passada é considerada.

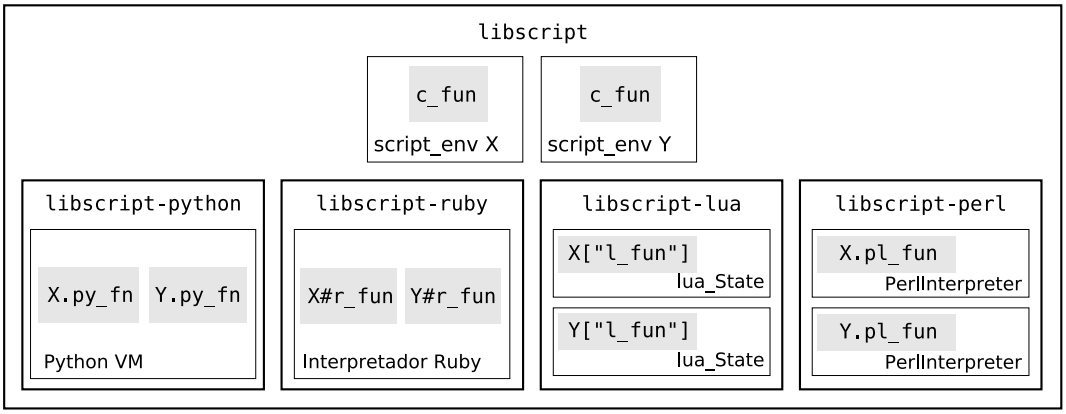

Figura 4.2: Ambientes virtuais em LibScript

ambientes, a aplicação registrou uma função C com o nome `c_fun` (que podem ou não corresponder à mesma função C). Scripts foram executados nestes ambientes, o que causou a carga dos plugins. No exemplo, estes scripts registraram algumas funções (`X.py_fn`, `Y.py_fn`, `X#r_fun`, etc.).

À parte da função para criação de ambientes virtuais, todas as demais funções da API de LibScript recebem como parâmetro um ambiente virtual sobre a qual elas devem operar. Isto indica em qual estrutura de C devem ser armazenadas mensagens de erro e valores de retorno. No caso de linguagens que permitem múltiplos estados de execução independentes, como Lua e Perl, isto indica também em qual estado o script deve executar.

Quando um script declara uma função no ambiente virtual, esta função passa a ser acessível através da API de LibScript. Por exemplo, no plugin Lua, o ambiente virtual é representado por uma tabela com o nome do ambiente; uma vez que um método Ruby `r` é declarado na classe `X`, esta função passa a poder ser invocada por C (através da API de LibScript) ou pelos outros plugins. Assim, por exemplo, embora na tabela Lua que implementa o ambiente virtual `X` só conste a função `l_fun`, scripts Lua podem invocar as demais funções através do ambiente virtual, como `X.c_fun` e `X.r_fun`. Estas chamadas serão tratadas pela biblioteca principal e resolvidas por ela, no caso de funções C como `X.c_fun`, ou repassadas para o plugin apropriado, como no caso de `X.r_fun`, realizando a chamada no plugin Ruby e passando os valores de retorno para o plugin Lua. A biblioteca principal localiza a função a ser executada consultando os plugins, conforme será explicado na Seção 4.1.3.

Na implementação dos plugins, utilizamos recursos oferecidos pelas linguagens para tratar acessos a elementos inexistentes nas estruturas, capturando estes acessos e repassando-os para a biblioteca principal. Estes recursos serão discutidos na Seção 4.2.4.

4.1.2 A API da biblioteca principal

A API oferecida por LibScript isola a aplicação das diferentes APIs oferecidas pelas linguagens de script. Não se trata apenas de adicionar uma camada de indireção entre as chamadas, o que seria apropriado apenas para os recursos que são comuns a todas elas, como inicialização e chamadas de função. A questão principal aí são os vários recursos particulares a cada linguagem. Uma abordagem pouco prática seria definir a API como a união dos conjuntos de recursos de todas as linguagens a ser suportadas (oferecer recursos de manipulação de seqüências para mapear este recurso de Python, recursos de manipulação de tabelas para Lua, e assim por diante). Este caminho traria vários problemas: a API seria complexa e provavelmente precisaria ser estendida a cada nova linguagem introduzida; mesmo para mapeamentos que aparentemente poderiam ser reaproveitados (por exemplo, mapear *hashes* de Python e tabelas de Lua para uma mesma API de *arrays* associativos) há o problema de sutis variações de semântica entre as implementações dos recursos nas diferentes linguagens. Além disso, bindings de aplicações poderiam oferecer funcionalidades disponíveis apenas para uma linguagem, indo contra a proposta de independência de linguagem de LibScript.

Outra abordagem é, ao invés de expor a API da linguagem à aplicação, expor apenas uma API de funções da aplicação para a linguagem e manter as estruturas de dados e recursos desta restrito ao domínio que será invocado. A aplicação interage com a máquina virtual enviando strings de código a ser executado e obtém resultados de volta quando o script passa parâmetros ao chamar funções da aplicação. Esta abordagem é proposta em (Thomas 2002) e utiliza o que, por exemplo, Python chama de "very high level layer" (van Rossum 2006a, van Rossum 2006b). Oferecer uma primitiva para a execução de uma string de código é algo básico em linguagens voltadas a script – `luaL_loadstring` em Lua, `PyRun_SimpleString` em Python, `rb_eval_string` em Ruby , `perl_eval_sv` em Perl (MacEachern 2006).

LibScript adota esta abordagem mais minimalista para sua API: não são oferecidas operações específicas para manipulação de estruturas de dados, apenas para a *execução de strings* – `script_run` (e a função de conveniência `script_run_file`, que lê um arquivo e o envia para `script_run`) – e *chamadas de função* com tipos básicos (números e strings) – `script_call`. Operações sobre dados mais complexos de tipos específicos de cada linguagem, quando necessárias, podem ser encapsuladas em funções implementadas nas linguagens de script. Pode-se ainda referenciar objetos da linguagem a partir de C armazenando-as em estruturas na linguagem de script e retornando a C índices

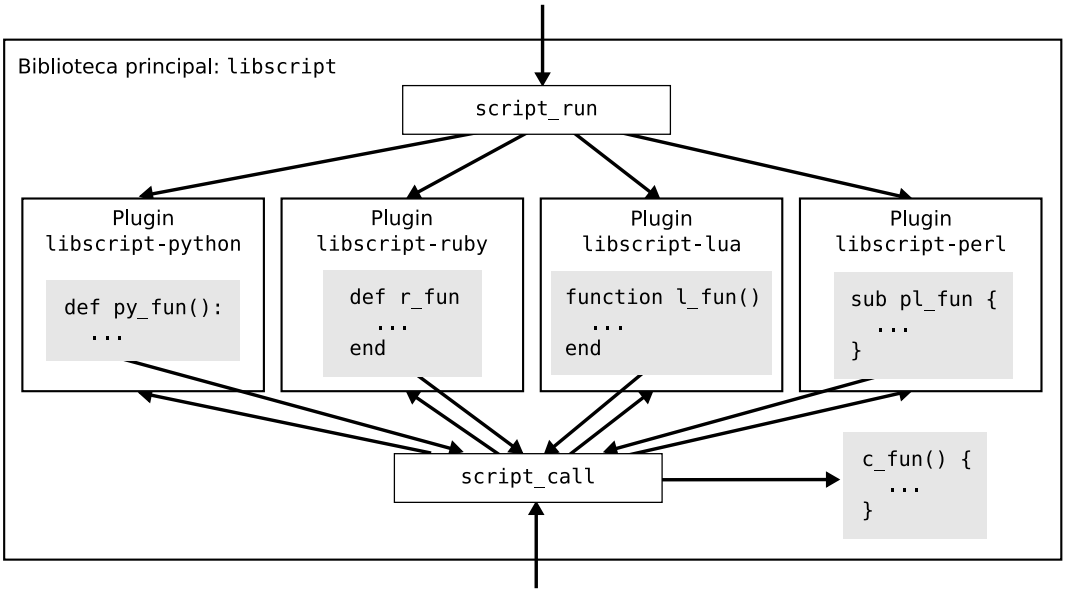

Figura 4.3: API para execução de código em LibScript

numéricos destas estruturas, servindo como *handles* de alto nível para os objetos.

A Figura 4.3 ilustra a interação entre a aplicação, a biblioteca principal e os plugins em relação a estas duas operações fundamentais, simbolizadas pelas funções `script_run` e `script_call`. Para a execução de strings, a biblioteca principal recebe da aplicação a entrada e repassa o código a ser executado para o plugin apropriado. Em `script_run`, são passadas duas strings, uma identificando a linguagem e outra contendo o código; em `script_run_file`, um nome de arquivo[2]. O exemplo a seguir declara um ambiente virtual, registra uma função C chamada `hello` e a invoca a partir de código Lua:

```
script_env* env = script_init("exemplo");
script_new_function(env, hello, "hello");
script_run(env, "lua", "exemplo.hello()");
```

O ambiente virtual é declarado com a função `script_init`, que recebe o nome que identificará o ambiente e retorna um identificador do tipo `script_env`, um ponteiro opaco que representa um ambiente virtual. A função C é registrada usando a função `script_new_function`, que recebe como parâmetros o ambiente, a função a ser registrada e o nome que a função terá no ambiente virtual. No código Lua, a função é acessada como um elemento `hello` (nome registrado da função) da tabela global `exemplo` (nome do ambiente virtual).

[2]Para código executado com `script_run_file`, a linguagem é automaticamente detectada como discutido na seção anterior.

Para a chamada de funções, a aplicação deve passar os parâmetros de entrada (a forma será discutida mais adiante), e chamar `script_call`, indicando o nome de uma função registrada no ambiente virtual. A mesma função `script_call` é usada pelos plugins quando eles desejam invocar funções do ambiente virtual registradas em C ou implementadas por outros plugins.

Por este motivo, procuramos usar uma API para transferência de dados genérica, a ser usada tanto na entrada como na saída de dados, tanto na comunicação entre a aplicação e a biblioteca principal como entre a biblioteca principal e os plugins. Optamos por uma abordagem similar às empregadas em Lua (Seção 3.3.4) e Perl (Seção 3.3.5) para o envio de dados na passagem de parâmetros e obtenção de valores de retorno, usando um buffer interno como área de transferência. Diferentemente destas linguagens, entretanto, passamos índices para os parâmetros explicitamente ao invés de empregar uma disciplina de pilha. As funções `script_{get,put}_{string,int,double,bool}` são usadas na entrada e saída de valores. As funções `script_put_*` armazenam valores no buffer interno e `script_get_*` os removem. Uma chamada a uma função `teste` passando uma string e um inteiro como parâmetros e obtendo um inteiro como resultado é realizada da seguinte forma:

```
script_put_string(env, 0, "entrada"); /* índice 0: "entrada" */
script_put_int(env, 1, 2);            /* índice 1: 2 */
script_call(env, "teste");
resultado = script_get_int(env, 0);   /* retorna índice 0 */
```

Chamadas de função são disponibilizadas como uma operação primitiva pois elas permitem um grau mínimo de interoperabilidade de forma independente de linguagem. Dois objetivos são atingidos desta forma. O primeiro é que assim programas C embutindo LibScript podem acessar a funcionalidade de scripts carregados sem precisar incluir no seu código strings de texto em alguma linguagem específica, por exemplo, inserindo em seu código uma chamada a uma função de *callback* a ser definida via script. Note que no exemplo acima, não é especificada a linguagem em que a função `teste` é implementada. Se a chamada fosse feita via execução de string de código, isto atrelaria a aplicação a pelo menos uma linguagem de script. Usando `script_run_file` e `script_call`, pode-se implementar uma aplicação extensível sem especificar explicitamente a linguagem de script a ser usada. O segundo objetivo é permitir que os próprios plugins possam invocar funções definidas em outros plugins. De qualquer forma, necessariamente teríamos que prover aos plugins uma função de chamada, para que eles pudessem invocar as funções C registradas em LibScript. Tornar a função de chamada genérica o suficiente para que

possa invocar também funções implementadas nos próprios plugins não torna, então, a API da biblioteca principal mais complexa.

O buffer de LibScript foi projetado para ser usado apenas como uma área de transferência temporária entre a biblioteca principal e os plugins, e não como uma facilidade geral para armazenamento e manipulação de dados. Assim, a sua API é voltada para a inserção e remoção seqüencial de elementos. Por exemplo, a inserção de um elemento na posição 0 automaticamente zera o buffer, evitando em muitos casos a necessidade de usar a função `script_reset_buffer`, que realiza tal operação explicitamente.

Funções C registradas com `script_new_function` devem receber o ambiente virtual como parâmetro e retornar um código de erro. As funções `script_get_*` e `script_put_*` são usadas para receber parâmetros e retornar valores ao implementar funções que podem ser chamadas via LibScript, da mesma forma que são usadas para passar parâmetros e obter valores de retorno ao realizar chamadas com `script_call`.

```
script_err teste_lua(script_env* env) {
   /* Entrada, índice 0: string */
   char* entrada = script_get_string(env, 0);
   /* Entrada, índice 1: inteiro */
   int n = script_get_int(env, 1);
   /* Sai da função com erro se algum script_get_* falhou */
   SCRIPT_CHECK_INPUTS(env);
   printf("Recebi %s e %ld \n", entrada, n);
   free(entrada);
   /* Retorno, índice 0: inteiro */
   script_put_int(env, 0, 42);
   return SCRIPT_OK;
}
```

Em LibScript as strings retornadas por `script_get_string` pertencem à função chamadora, sendo responsabilidade dela desalocar a memória, diferentemente do que ocorre nas funções similares das APIs das linguagens discutidas neste trabalho. Tal decisão foi tomada devido ao caráter temporário do buffer de LibScript: retornar ao chamador um ponteiro para uma string cuja validade seria restrita até a próxima chamada da API seria algo pouco intuitivo, e na prática forçaria freqüentemente o programador a fazer uma cópia da string.

4.1.3 A API de plugins

Um plugin que embute uma linguagem de script deve implementar quatro operações: `init`, `run`, `call` e `done`. A biblioteca principal espera que a

biblioteca dinâmica que implementa o plugin de uma linguagem exporte quatro funções, com nomes do tipo `script_plugin_`*[operação]*`_`*[linguagem]*.

A função `script_plugin_init_`*[linguagem]* é responsável pela inicialização de um plugin, e é chamada pela função `script_init` da biblioteca principal. Na inicialização de um plugin, a biblioteca principal passa à função `script_plugin_init_`*[linguagem]* um ponteiro `script_env` e recebe um `script_plugin_state`, que é um tipo opaco que é sempre passado de volta ao plugin nas demais chamadas. Cada plugin define a sua representação interna para `script_plugin_state`. Tipicamente, o estado da máquina virtual e o ponteiro para o ambiente LibScript devem ser armazenados de modo a ser posteriormente acessíveis a partir deste handle. Na Seção 4.2.1 discutiremos como cada plugin representa o ambiente e o seu estado interno em `script_plugin_state`.

A função `script_plugin_run_`*[linguagem]* é invocada por `script_run`. Ela recebe uma string contendo código da linguagem de script, executa este código na máquina virtual e retorna um valor de status indicando sucesso ou a ocorrência de erros de compilação ou execução. No caso de erros, os plugins devem capturar exceções disparadas pela máquina virtual e retornar a constante `SCRIPT_ERRLANGRUN`. Caso seja possível obter da linguagem uma mensagem de erro, esta pode ser propagada usando a função `script_set_error_message` da biblioteca principal. A mensagem armazenada por ela poderá ser posteriormente consultada pela aplicação com a função `script_error_message`.

A função `script_plugin_call_`*[linguagem]* é usada por `script_call`, e é responsável por realizar chamadas a funções implementadas na linguagem embutida pelo plugin. Se a função foi definida no plugin, isto é, se uma função com o nome solicitado foi registrada na estrutura de dados que descreve o ambiente na máquina virtual, ela será executada, e o sucesso ou falha da execução será reportado de forma igual a `script_plugin_run_`*[linguagem]*. Caso a função solicitada não tenha sido definida na máquina virtual, `script_plugin_call_`*[linguagem]* deve retornar a constante `SCRIPT_ERRFNUNDEF`. Parâmetros de entrada e valores de retorno são passados através do buffer de parâmetros, usando as mesmas funções `script_get_*` e `script_put_*` da biblioteca principal que são usadas para a passagem de dados entre a aplicação e a biblioteca principal.

A implementação da função `script_call` na biblioteca principal faz uso deste comportamento dos plugins para invocar funções de modo independente de linguagem. Inicialmente, ela tenta encontrar uma função solicitada na lista de funções C registradas. Caso não haja uma função C no ambiente virtual com este nome, `script_call` tenta localizar a função nos plugins carregados, chamando a função com `script_plugin_call_`*[linguagem]* em cada plugin, e

tentando o próximo a cada vez que recebe `SCRIPT_ERRFNUNDEF`.

Finalmente, a função `script_plugin_done_`*[linguagem]* é chamada por `script_done` quando um ambiente virtual é encerrado. Dependendo da representação interna usada no plugin, a finalização de um estado pode ou não implicar na finalização da máquina virtual. Preferencialmente, esta função deve remover a estrutura que descreve o ambiente virtual, mas, como veremos na Seção 4.2.2, isto nem sempre é possível.

4.2 Implementação dos plugins

Nesta seção discutiremos os principais aspectos envolvidos na implementação dos plugins desenvolvidos neste estudo de caso. Implementamos plugins para as linguagens Python, Ruby, Lua e Perl. Apresentaremos aqui como é feita a representação dos estados virtuais em cada plugin (Seção 4.2.1), questões envolvendo o encerramento de estados (Seção 4.2.2), passagem de parâmetros entre a biblioteca principal e os plugins (Seção 4.2.3), como a chamada de funções a partir de scripts é tratada pelos plugins (Seção 4.2.4) e a captura de erros (Seção 4.2.5).

4.2.1 Representação de estados

O design de LibScript permite que plugins mantenham múltiplos estados de execução independentes. Idealmente estes estados seriam totalmente isolados entre si, como por exemplo diferentes instâncias da máquina virtual, oferecendo maior segurança ao ambiente de execução dos scripts. Todavia, as linguagens oferecem diferentes graus de isolamento possível entre estados independentes. Lua e Perl permitem múltiplas instâncias isoladas do ambiente de execução de forma simples, uma vez que as chamadas à API incluem um identificador de estado[3]. Já linguagens que mantêm estado de forma estática, como Python e Ruby, não permitem trabalhar com múltiplos estados isolados facilmente[4]. Nas linguagens que não permitem múltiplas instâncias da máquina virtual, podemos definir apenas espaços de nomes separados para os ambientes virtuais LibScript, que compartilham um único estado global de execução

[3]O recurso de múltiplos estados independentes é opcional em Perl, selecionado durante a compilação da biblioteca do interpretador.

[4]O modelo de threads de Python oferece uma forma de alternar entre estados na máquina virtual obtendo objetos `PyThreadState` através da chamada `Py_NewInterpreter`, mas isto pode causar problemas quando módulos de extensão escritos em C utilizam variáveis globais estáticas ou quando módulos manipulam o seu próprio dicionário, que é compartilhado entre estados. A documentação diz, desde 1999, que "*This is a hard-to-fix bug that will be addressed in a future release.*" (van Rossum 1999, van Rossum 2006c)

dentro do plugin. À representação de um estado de execução relativo a um ambiente virtual LibScript dentro de um plugin damos o nome de *estado virtual*, que pode ou não corresponder a um estado de execução isolado.

Como comentado na seção anterior, a função `script_plugin_init_`*linguagem* retorna à biblioteca principal um `script_plugin_state`, que é a representação opaca do seu estado virtual. O conteúdo desta representação varia de linguagem para linguagem, mas o princípio básico é que dois dados devem estar disponíveis a partir deste valor: uma referência para o ambiente virtual LibScript, recebido como parâmetro para `script_plugin_init_`*linguagem*, para que o plugin possa fazer chamadas à biblioteca principal, e um identificador que permita ao plugin acessar a estrutura de dados que representa na linguagem o espaço de nome de funções acessíveis via LibScript. No plugin Lua, esta estrutura é uma tabela; em Python, um módulo; em Ruby, uma classe; e em Perl, um pacote.

Em LibScript-Lua, estados são implementados como `lua_State`s (Seção 3.4.4). Desta forma, scripts executados em um ambiente são plenamente isolados dos demais ambientes. Por exemplo, a alteração do valor de uma variável global em um ambiente não afeta os demais. De fato, o `script_plugin_state` retornado pelo plugin Lua é simplesmente o `lua_State` convertido via cast. O ponteiro para o ambiente LibScript é armazenado em Lua no registro, da seguinte forma:

```
/* Empilha o índice */
lua_pushstring(L, "LibScript.env");
/* Empilha o ambiente LibScript */
lua_pushlightuserdata(L, env);
/* registro["LibScript.env"] = env */
lua_settable(L, LUA_REGISTRYINDEX);
```

O plugin cria neste `lua_State` uma tabela que representará o ambiente virtual para scripts Lua. Esta tabela é armazenada no `lua_State` como uma variável global com o nome do ambiente virtual.

Em LibScript-Perl os estados são isolados como em Lua. Cada estado criado inicializa uma nova instância de `PerlInterpreter`. Neste interpretador, é criado um pacote que será a representação do ambiente visível a partir de código Perl. O tipo `script_plugin_state`, então, é um *typedef* para `PerlInterpreter`*.

Como discutido na Seção 3.4.5, a implementação de funções C exportadas para um interpretador Perl é feita escrevendo um módulo de extensão usando o pré-processador XS, e a forma de obter comunicação no sentido Perl→C em uma máquina virtual embutida é ligando um módulo de extensão juntamente

com a máquina virtual. Assim, parte do plugin LibScript-Perl é implementado como um módulo XS, exposto na máquina virtual embutida como o pacote Perl `LibScript`. Durante a inicialização de um estado virtual, o ponteiro para o ambiente virtual LibScript é armazenado neste pacote, na variável `$LibScript::env`. O pacote que representa o ambiente virtual é criado pela função `script_plugin_init_perl`, executando a string de código `"package` *[ambiente]*`;"` com a função `Perl_eval_pv`.

Como Python não dispõe de facilidades para disparar múltiplas máquinas virtuais plenamente isoladas, o plugin Python implementa estados virtuais apenas como módulos separados, compartilhando um mesmo estado global. Durante a inicialização de um estado, é criado um módulo Python com o nome do ambiente. O seguinte trecho da função `script_plugin_init_python` exibe a seqüência onde o módulo é criado e importado:

```
/* Obtém o nome do ambiente */
char* namespace = script_namespace(env);
/* Cria o módulo. O primeiro parâmetro é o nome do módulo, o segundo
   a lista de métodos do módulo, que será inicialmente vazio. */
PyObject* module = Py_InitModule3(namespace, NULL);
/* Obtém dicionário de globais */
PyObject* globals = PyModule_GetDict(PyImport_AddModule("__builtin__"));
/* Atribui o módulo à variável global com o seu nome. */
PyDict_SetItemString(globals, namespace, module);
```

O tipo `script_plugin_state` é um *typedef* para `PyObject*`. O objeto retornado pela função de inicialização é o dicionário de elementos do módulo, obtido com `PyModule_GetDict(module)`. Neste dicionário, armazenamos o ponteiro do ambiente virtual como o atributo privado `__env`.

De forma similar, em Ruby estados virtuais são implementados como classes que compartilham um mesmo estado global, já que Ruby também não permite múltiplos ambientes de execução isolados. Na função de inicialização `script_plugin_init_ruby`, uma classe com o nome do ambiente virtual é criada usando a função `rb_define_class`. O ponteiro do ambiente virtual é armazenado em uma constante da classe como um número. O `VALUE` referente à classe é retornado como o `script_plugin_state`.

```
VALUE state;
/* ... (inicialização do interpretador omitida) ... */
/* class_name é o nome do ambiente virtual,
   com a inicial convertida para maiúsculas,
   respeitando a convenção de nomes de classe Ruby */
state = rb_define_class(class_name, rb_cObject);
/* Isto assume que void* cabe em um long */
```

```
rb_const_set(state, rb_intern("@@LibScriptEnv"), INT2NUM((long)env));
/* ... */
return (script_plugin_state) state;
```

4.2.2 Encerramento de estados

Como Lua e Perl representam estados de forma independente, o encerramento de um estado nestes plugins é simples: a estrutura da linguagem que encapsula o ambiente completo de execução é encerrada. A implementação da função de finalização no plugin Lua é a seguinte:

```
void script_plugin_done_lua(script_plugin_state state) {
   /* Em Lua, um state é um lua_State */
   lua_State* L = (lua_State*) state;
   /* Encerra o estado. Não afeta outros ambientes. */
   lua_close(L);
}
```

Em Perl, o processo, embora um tanto mais elaborado, é essencialmente similar:

```
void script_plugin_done_perl(script_perl_state* state) {
   /* Algumas macros assumem que o ponteiro do interpretador
      se chama my_perl. */
   PerlInterpreter* my_perl = (PerlInterpreter*) state;
   /* Algumas operações atuam sobre o ''estado atual'',
      então a macro PERL_SET_CONTEXT deve ser usada para
      alternar o interpretador ativo */
   PERL_SET_CONTEXT(my_perl);
   /* Esta flag deve ser ativada para que a limpeza do
      ambiente seja completa, o que é necessário quando
      pode haver mais de um interpretador ativo */
   PL_perl_destruct_level = 1;
   /* Encerramento do interpretador */
   perl_destruct(my_perl);
   perl_free(my_perl);
}
```

Em Python e Ruby, o plugin precisa manter o controle do número de estados ativos para desalocar a máquina virtual somente quando este chegar a zero. Além disso, tanto em Ruby como em Python não há recursos nas APIs (ou nas linguagens, de fato) para remover, respectivamente, classes ou módulos. Em Ruby, poderíamos atribuir `nil` à constante que representa a classe que descreve o ambiente virtual, mas depois disso não é possível definir uma nova classe em seu lugar: tanto `rb_define_class` via C como `class` *[Nome]* via

Ruby geram um erro indicando que o valor já foi definido com outro tipo. Como Ruby possui classes abertas, uma construção `class` *[Nome]* para um *[Nome]* já existente é entendida como a continuação da descrição da classe, e não como a redefinição de *[Nome]*. Python, por sua vez, não disponibiliza recursos na API para a descarga de módulos, mas permite atribuir `None` à global referente ao módulo. O módulo pode ser importado novamente, mas a mesma instância dele, armazenada internamente por Python, será retornada. A seguinte sessão interativa de linha de comando permite observar este comportamento, que ocorre tanto diretamente em Python como via a API de C:

```
>>> import sys
>>> sys.foo = "hello"
>>> sys.foo
'hello'
>>> sys = None
>>> import sys
>>> sys.foo
'hello'
```

Assim, as estruturas de dados referentes aos estados LibScript não são encerrados nos plugins Python e Ruby. Esta é a implementação da rotina de encerramento no plugin Ruby:

```
void script_plugin_done_ruby(script_ruby_state state) {
   /* Decrementa o contador de estados,
      uma variável global static do plugin. */
   script_ruby_state_count--;
   /* Finaliza o interpretador se este for o último estado. */
   if (script_ruby_state_count == 0)
      ruby_finalize();
}
```

A implementação no plugin Python é basicamente igual:

```
void script_plugin_done_python(script_python_state state) {
   script_python_state_count--;
   if (script_python_state_count == 0)
      Py_Finalize();
}
```

4.2.3 Passagem de parâmetros

A transferência de dados entre a biblioteca principal e os plugins é concentrada em duas operações: uma para passar o conteúdo do buffer de parâmetros de LibScript para o espaço de dados da máquina virtual e outra para realizar a operação inversa. A primeira é usada na passagem de parâmetros de entrada quando funções da linguagem de script são chamadas por C e para a obtenção dos valores de retorno quando a linguagem de script faz chamadas que são tratadas por C. A segunda operação, de forma complementar, é usada para os valores de retorno quando C chama a linguagem de script e para os parâmetros de entrada quando uma chamada feita pela linguagem de script é tratada por código C.

Na implementação do plugin LibScript-Lua, a função `script_lua_stack_to_buffer` converte o conteúdo da pilha de Lua para o buffer de parâmetros de LibScript. A função do plugin responsável por invocar funções Lua a partir de C, `script_plugin_call_lua`, usa a função `script_lua_stack_to_buffer` para armazenar no buffer LibScript os valores de retorno da função Lua invocada, já que estes são retornados na pilha virtual. Quando o código Lua chama funções implementadas em C ou em outro plugin, `script_lua_stack_to_buffer` é usada para converter os parâmetros de entrada da função, também recebidos na pilha virtual. A seguir, vemos a implementação desta função:

```
static void script_lua_stack_to_buffer(script_env* env, lua_State *L) {
   int nargs; int i;
   /* Número de elementos na pilha de Lua */
   nargs = lua_gettop(L);
   script_reset_buffer(env); /* Esvazia o buffer LibScript */
   for (i = 1; i <= nargs; i++) {
      /* Verifica o tipo Lua do elemento na posição i da pilha
      e para cada tipo, converte o elemento e o armazena no buffer */
      switch(lua_type(L, i)) {
      case LUA_TNUMBER:
         script_put_double(env, i-1, lua_tonumber(L, i)); break;
      case LUA_TSTRING:
         script_put_string(env, i-1, lua_tostring(L, i)); break;
      case LUA_TBOOLEAN:
         script_put_bool(env, i-1, lua_toboolean(L, i)); break;
      default:
         /* Tipos não tratados são substituídos por zero. */
         script_put_double(env, i-1, 0);
      }
```

```
    }
}
```

Assumimos em LibScript strings no formato de C: a função `script_put_string` copia a string passada até o primeiro `'\0'`. Assim, ao obter strings de linguagens que permitem conteúdo arbitrário, estas serão truncadas caso contenham `'\0'`. Por isso, no plugin Lua usamos diretamente a função `lua_tostring`, e não a função mais geral `lua_tolstring` (que retorna também o tamanho do buffer). Esta decisão de projeto coincide com o objetivo explicado anteriormente de restringirmos a API da biblioteca principal a recursos disponíveis em todas as linguagens.

Os valores de tipos desconhecidos são substituídos pelo valor zero, o que mantém a posição dos demais valores na lista de argumentos. Optamos por não sinalizar erro nesta situação para evitar aqui a geração de exceções, o que complicaria a exposição. A captura e propagação de erros serão vistas na Seção 4.2.5.

A segunda função de transferência de dados de LibScript-Lua, `script_lua_buffer_to_stack`, obtém os valores do buffer LibScript e os insere na pilha virtual de Lua. Esta função é usada para passar os parâmetros de entrada para Lua em `script_plugin_call_lua` e para passar para Lua os valores obtidos pelo retorno da função `script_call`, que é invocada internamente pelo plugin quando Lua invoca uma função C.

```
static int script_lua_buffer_to_stack(script_env* env, lua_State *L) {
   int i; char* s;
   /* Número de elementos no buffer */
   int len = script_buffer_len(env);
   for (i = 0; i < len; i++) {
      /* Verifica o tipo do elemento na posição i do buffer */
      /* e para cada tipo, o obtém e o insere na pilha de Lua */
      type = script_get_type(env, i);
      switch (type) {
      case SCRIPT_DOUBLE:
         lua_pushnumber(L, script_get_double(env, i)); break;
      case SCRIPT_STRING:
         /* A string pertence ao chamador. */
         s = script_get_string(env, i);
         lua_pushstring(L, s);
         /* Libera a string,
            já que Lua armazena sua própria cópia. */
         free(s);
         break;
      case SCRIPT_BOOL:
```

```
            lua_pushboolean(L, script_get_bool(env, i)); break;
         }
      }
      return len;
   }
```

Em LibScript-Python, não foi possível concentrar as operações de transferência de dados em apenas duas funções. Cada operação teve que ser dividida em duas partes. A conversão de dados enviados de Python para o buffer de LibScript foi divida nas funções `script_python_put_object` e `script_python_tuple_to_buffer`. A primeira função converte um único valor Python e o insere na posição solicitada no buffer:

```
static void script_python_put_object(script_env* env, int i, PyObject* o) {
   if (PyString_Check(o))
      script_put_string(env, i, PyString_AS_STRING(o));
   else if (PyInt_Check(o))
      script_put_int(env, i, PyInt_AS_LONG(o));
   else if (PyLong_Check(o))
      script_put_double(env, i, PyLong_AsDouble(o));
   else if (PyFloat_Check(o))
      script_put_double(env, i, PyFloat_AS_DOUBLE(o));
   else if (PyBool_Check(o))
      script_put_bool(env, i, o == Py_True ? 1 : 0);
   else
      script_put_int(env, i, 0);
}
```

É importante notar que os tipos Python `PyInt` e `PyLong` não correspondem aos tipos C `int` e `long`: `PyInt` é o tipo inteiro correspondente ao tamanho da palavra da máquina (análogo a `int`), mas `PyLong` é um inteiro de precisão arbitrária. Em LibScript, representamos `PyLong`s como `double`s. A API de LibScript oferece a função `script_put_int` como conveniência, mas internamente, como ocorre por exemplo em Lua, todos os números são armazenados como `double`s.

A segunda função, `script_python_tuple_to_buffer`, insere os elementos de uma tupla no buffer:

```
static void script_python_tuple_to_buffer(script_env* env, PyObject* tuple) {
   int i;
   /* Número de elementos da tupla */
   int len = PyTuple_GET_SIZE(tuple);
   /* Esvazia o buffer LibScript */
```

```
    script_reset_buffer(env);
    for (i = 0; i < len; i++) {
      /* Obtém elemento da tupla */
      PyObject* o = PyTuple_GET_ITEM(tuple, i);
      /* Insere-o no buffer. */
      script_python_put_object(env, i, o);
    }
}
```

A operação inversa, de transferência de dados do buffer LibScript para Python, também é implementada em duas funções, uma tratando objetos individualmente e outra tratando tuplas. A função `script_get_object` converte um elemento do buffer para um `PyObject` equivalente:

```
static PyObject* script_python_get_object(script_env* env, int i) {
    PyObject* ret; char* s;
    switch (script_get_type(env, i)) {
    case SCRIPT_DOUBLE:
      return PyFloat_FromDouble(script_get_double(env, i));
    case SCRIPT_STRING:
      s = script_get_string(env, i);
      PyObject* ret = PyString_FromString(s);
      free(s);
      return ret;
    case SCRIPT_BOOL:
      return PyBool_FromLong(script_get_bool(env, i));
    }
}
```

A função `script_python_buffer_to_tuple` gera uma tupla contendo todos os elementos do buffer LibScript:

```
static PyObject* script_python_buffer_to_tuple(script_env* env) {
    int i;
    int len = script_buffer_len(env);
    PyObject* ret = PyTuple_New(len);
    for(i = 0; i < len; i++) {
      PyObject* o = script_python_get_object(env, i);
      PyTuple_SetItem(ret, i, o);
    }
    return ret;
}
```

Assim, estes dois pares de funções realizam funções equivalentes às que `script_lua_stack_to_buffer` e `script_lua_buffer_to_stack` exercem

no plugin Lua. Elas foram separadas em duas partes em função do modelo de retorno de valores em funções Python: no caso de múltiplos valores de retorno, eles são retornados como uma tupla; para valores simples, eles são passados diretamente. Isto é evidenciado no seguinte trecho da função `script_plugin_call_python`:

```
PyObject *ret, *args;
/* ... */
/* Obtém o parâmetros de entrada */
args = script_python_buffer_to_tuple(env);
/* Chama uma função Python */
ret = PyEval_CallObject(func, args);
/* ... */
/* Se a função não retornou valor */
if (ret == Py_None)
   /* Apenas zere o buffer LibScript */
   script_reset_buffer(env);
/* Se retornou uma tupla */
else if (PyTuple_Check(ret))
   /* Insira seus elementos no buffer */
   script_python_tuple_to_buffer(env, ret);
/* Se retornou outro tipo de objeto */
else
   /* Insira-o como único elemento */
   script_python_put_object(env, 0, ret);
```

No tratador de chamadas a funções externas do plugin, a comunicação no sentido inverso emprega uma lógica similar:

```
/* Obtém o parâmetros de entrada */
script_python_tuple_to_buffer(env, args);
/* Chama um função via LibScript */
err = script_call(env, fn_name);
/* ... */
switch(script_buffer_len(env)) {
/* Se a função não retornou valor */
case 0:
   /* Retorne o valor Python 'None' */
   Py_RETURN_NONE;
/* Se retornou um único valor */
case 1:
   /* Converta e retorne-o */
   return script_python_get_object(env, 0);
/* Se retornou mais de um valor */
default:
   /* Retorne-os em uma tupla */
   return script_python_buffer_to_tuple(env);
}
```

Assim como em Python, funções em Ruby retornam múltiplos valores encapsulando-os em um tipo agregado. Desta forma, as operações de transferências de dados de LibScript-Ruby também são divididas em pares de funções, uma convertendo um valor do buffer, e outra operando sobre um array Ruby. A função análoga a `script_python_put_object` é `script_ruby_put_value`:

```
static void script_ruby_put_value(script_env* env, int i, VALUE arg) {
   switch (TYPE(arg)) {
   case T_FLOAT:
   case T_FIXNUM:
   case T_BIGNUM:
      script_put_double(env, i, NUM2DBL(arg)); break;
   case T_STRING:
      script_put_string(env, i, StringValuePtr(arg)); break;
   case T_TRUE:
      script_put_bool(env, i, 1); break;
   case T_FALSE:
      script_put_bool(env, i, 0); break;
   default:
      script_put_int(env, i, 0);
   }
}
```

Aqui, alguns problemas da API de Ruby são aparentes. Além da inconsistência na nomenclatura das funções de conversão de objetos, o significado do valor retornado pela macro `TYPE` só pode ser compreendido através da representação interna de `VALUE`s na implementação de Ruby, e não através da hierarquia de tipos dos objetos da linguagem. As classes que têm tratamento especial na estrutura interna de `VALUE`s possuem constantes associadas a si, como `T_FLOAT` e `T_STRING`; as demais são identificados apenas como `T_OBJECT`s. O uso de `T_TRUE` e `T_FALSE` pode dar a entender que alguns valores específicos também retornam resultados especiais para `TYPE`. De fato, estes valores são definidos como `VALUE`s que não correspondem a índices da heap de objetos de Ruby e são tratados de forma especial na implementação. Do ponto de vista de código Ruby, entretanto, esta classificação dos valores `true` e `false` em tipos separados na API C é justificada definindo-os como *singletons* das classes `TrueClass` e `FalseClass`, abordagem provavelmente influenciada por Smalltalk. Porém, diferentemente de Smalltalk, onde `True` e `False` são subclasses de `Boolean`, em Ruby `TrueClass` e `FalseClass` são subclasses diretas de `Object`. Isto traz a inconveniência de que verificar se um tipo é um valor booleano incorre sempre em dois testes.

Assim como LibScript-Python tem uma função para armazenar no buffer os elementos de uma tupla, LibScript-Ruby possui uma função para armazenar os elementos de um array:

```
static void script_ruby_array_to_buffer(script_env* env, VALUE array) {
   int i;
   int len = RARRAY(array)->len;
   script_reset_buffer(env);
   for (i = 0; i < len; i++) {
      VALUE o = rb_ary_entry(array, i);
      script_ruby_put_value(env, i, o);
   }
}
```

Ruby não possui uma função na API C para retornar o tamanho de um array; ao invés disso, a estrutura interna do `VALUE` é exposta através da macro `RARRAY` (que apenas encapsula um cast).

As operações para conversão de valores do buffer LibScript para Ruby também são similares às implementadas no plugin Python. Novamente, onde em Python há uma função para manipulação de tuplas, temos em Ruby uma função que opera sobre arrays:

```
static VALUE script_ruby_get_value(script_env* env, int i) {
   VALUE ret; char* s;
   switch (script_get_type(env, i)) {
   case SCRIPT_DOUBLE:
      return rb_float_new(script_get_double(env, i));
   case SCRIPT_STRING:
      s = script_get_string(env, i);
      ret = rb_str_new2(s);
      free(s);
      return ret;
   case SCRIPT_BOOL:
      return script_get_bool(env, i) ? Qtrue : Qfalse;
   }
}

static VALUE script_ruby_buffer_to_array(script_env* env) {
   int i;
   int len = script_buffer_len(env);
   VALUE ret = rb_ary_new2(len);
   for (i = 0; i < len; i++) {
      VALUE o = script_ruby_get_value(env, i);
      rb_ary_store(ret, i, o);
   }
```

```
    return ret;
}
```

De forma similar ao plugin Python, a implementação da chamada de funções Ruby a partir de LibScript usa a função `script_ruby_buffer_to_array` para converter os parâmetros de entrada e as funções `script_ruby_put_value` ou `script_ruby_array_to_buffer` para converter o valor de retorno, dependendo se a função retornou um ou mais valores (ou mais precisamente, se a função retornou ou não um array). Em chamadas de funções LibScript a partir de Ruby, os parâmetros de entrada são convertidos com `script_ruby_array_to_buffer` e os valores de retorno com `script_ruby_get_value` ou `script_ruby_buffer_to_array`.

No plugin Perl, temos três funções: a transferência de dados da pilha para o buffer LibScript pôde ser implementada em uma única função como em Lua, mas a transferência no sentido oposto teve que ser dividida em duas funções, como em Python e Ruby. Esta assimetria vem do fato de que o tratamento de valores de retorno é encapsulado pelo pré-processador XS através da variável especial `RETVAL`; assim, nesta situação não podemos manipular a pilha diretamente, mas apenas passar `SV`s como valores de saída.

A transferência de dados da pilha de Perl para o buffer LibScript é razoavelmente simples:

```
void script_perl_stack_to_buffer(pTHX_ int ax, script_env* env,
                                 int count, int offset) {
   int i;
   script_reset_buffer(env);
   for (i = 0; i < count; i++) {
      /* Obtém um ponteiro para o SV */
      SV* o = ST(offset+i);
      if (SvIOK(o))
         script_put_int(env, i, SvIV(o));
      else if (SvNOK(o))
         script_put_double(env, i, SvNV(o));
      else if (SvPOK(o))
         script_put_string(env, i, SvPV_nolen(o));
      else
         script_put_int(env, i, 0);
   }
}
```

Os parâmetros de entrada desta função merecem comentário. Inicialmente, temos a macro `pTHX_`. Esta macro foi adicionada à API quando Perl passou a permitir múltiplos interpretadores simultâneos por processo: as funções da API foram transformadas em macros que encapsulam a passagem deste

primeiro parâmetro. Por exemplo, a função `eval_sv` pode ser chamada como `Perl_eval_sv`, passando a macro `aTHX_` como parâmetro inicial. De maneira geral o uso destas macros fica implícito, mas ao escrever funções que usam a API de Perl torna-se necessário usar a macro `pTHX_` na declaração[5], para propagar a informação de estado do interpretador através de chamadas de função, e `aTHX_` nas chamadas.

Outro sintoma de que a API de Perl foi projetada mais para uso interno do pré-processador XS do que para manipulação direta transparece no segundo argumento, `ax`. Algumas macros assumem a existência deste valor, que não é propagado via `pTHX_`, mas é declarado implicitamente quando funções são encapsuladas via XS. A API parece assumir que uma função XS não irá invocar outra função C que também use a API. Tivemos então que propagar esta variável (que é citada na documentação, mas apenas como *"the 'ax' variable"* (Okamoto 2006a), sem explicações do seu propósito).

Os outros dois parâmetros, `count` e `offset`, são necessários devido às diferentes formas que as informações que eles representam são obtidas nos dois contextos onde esta função é usada. Nos outros plugins, podemos obter a quantidade de elementos de entrada de forma uniforme (consultando o número de elementos da tupla em Python, por exemplo). Em Perl, nas duas situações onde a função é chamada, o número de elementos a serem lidos da pilha deve ser obtido de formas diferentes, e por isso o passamos como parâmetro `count`. Na rotina chamadora de funções LibScript, implementada no arquivo XS, o tamanho da pilha é obtido através de uma variável especial, `items`. Já na chamada de funções Perl, o valor de `count` é obtido como retorno da função que realiza a invocação, `Perl_call_pv`.

A posição inicial da pilha a partir da qual devemos obter os elementos (`offset`) também varia. Dentro da função XS, os parâmetros de entrada começam a partir da posição 2, pois LibScript passa o ponteiro do ambiente e o nome da função nos dois primeiros argumentos. Na chamada de funções Perl, o valor de `offset` é zero pois, como visto no protocolo de chamada de funções Perl discutido na Seção 3.3.5, a base da pilha é ajustada após a chamada da função pela macro `SPAGAIN`.

A conversão de valores do buffer LibScript para a pilha de Perl é dada em duas funções, uma que gera um único `SV` e outra que empilha todos os elementos:

```
SV* script_perl_get_sv(pTHX_ script_env* env, int i) {
   switch (script_get_type(env, i)) {
```

[5]A macro `pTHX_` é usada sem a vírgula separando-a do argumento seguinte. Quando ela é o único argumento, deve-se usar `pTHX`.

```
    case SCRIPT_DOUBLE: return newSVnv(script_get_double(env, i));
    /* 0 indica que o tamanho da string deve ser calculado por Perl. */
    case SCRIPT_STRING: return newSVpv(script_get_string(env, i), 0);
    case SCRIPT_BOOL: return newSViv(script_get_bool(env, i));
    }
}

SV** script_perl_buffer_to_stack(pTHX_ SV** sp, script_env* env) {
    int i;
    int len = script_buffer_len(env);
    for (i = 0; i < len; i++) {
        XPUSHs(sv_2mortal(script_perl_get_sv(aTHX_ env, i)));
    }
    return sp;
}
```

Novamente, uma variável criada internamente por Perl teve que ser propagada explicitamente: `sp`, o *stack pointer*. Esta variável é referenciada internamente pela macro `xPUSHs`. Além disso, como `XPUSHs` pode redimensionar a pilha, precisamos retornar o valor atualizado de `sp` de volta para o chamador. No mais, a geração de `SV`s, o registro destes como variáveis mortais e o seu empilhamento ocorre da forma usual, já apresentada na Seção 3.3.5.

Assim como nos demais plugins, a passagem de parâmetros de entrada em LibScript-Perl, tanto para a chamada de funções Perl como de funções via LibScript, é feita chamando a função de conversão que opera sobre o buffer como um todo: na chamada de funções Perl usamos `script_perl_buffer_to_stack` e na de funções via LibScript, `script_perl_stack_to_buffer`. Para tratar os valores de retorno de funções Perl, pudemos utilizar diretamente a função `script_perl_stack_to_buffer`, de forma similar à realizada em LibScript-Lua. Para o retorno de funções chamadas via LibScript, porém, precisamos lidar com a variável especial `RETVAL` de XS e com os diferentes contextos de chamada de Perl. O trecho abaixo ilustra o tratamento de valores de retorno neste caso:

```
err = script_call(env, function_name);
/* ... (tratamento de erro omitido) ... */
switch (GIMME_V) {
case G_SCALAR:
    /* Retorna o primeiro item do buffer */
    RETVAL = script_perl_get_sv(aTHX_ env, 0);
    break;
case G_ARRAY:
    len = script_buffer_len(env);
    /* Cria um array */
```

```
        RETVAL = (SV*)newAV();
        /* Arrays retornados devem ser marcados como mortais */
        sv_2mortal((SV*)RETVAL);
        /* Insere o conteúdo do buffer no array */
        for (i = 0; i < len; i++)
            av_push((AV*)RETVAL, script_perl_get_sv(aTHX_ env, i));
        break;
    case G_VOID:
        /* O valor de retorno é descartado em contextos void. */
        /* Retornamos então a constante Perl undef. */
        RETVAL = &PL_sv_undef;
        break;
    }
```

4.2.4 Chamada de funções

Nos plugins de LibScript, funções implementadas externamente (em C ou outros plugins) são localizadas somente no momento em que elas são chamadas. O objetivo aqui, além de otimizar o tempo de inicialização e consumo de memória no ambiente de execução da linguagem de script (ao evitar a declaração de funções que não serão utilizadas), é permitir a localização de funções declaradas após a inicialização do ambiente. Para permitir esta resolução de funções de forma dinâmica, é preciso capturar o acesso a elementos inexistentes na estrutura que descreve o ambiente virtual no plugin e encaminhar a chamada à biblioteca principal via `script_call`. Ao comparar as abordagens empregadas em cada plugin para obter tal comportamento, podemos avaliar alguns recursos de meta-programação oferecidos por cada linguagem e a sua disponibilidade através das suas APIs.

Como vimos na Seção 4.2.1, em Lua, durante a inicialização do plugin, é criada uma tabela armazenada em uma variável global com o nome do ambiente. Funções são inseridas dinamicamente nesta tabela através da metatabela associada a ela logo após a sua criação em `script_plugin_init_lua`. O campo `__index` da metatabela aponta para uma função C interna ao plugin, `script_lua_make_caller`, que é então invocada sempre que um elemento inexistente for solicitado na tabela. A função `script_lua_make_caller` cria uma *closure* C, que consiste de outra função C interna ao plugin (`script_lua_caller`) e o nome da função solicitada. Esta closure é associada à entrada da tabela do ambiente. Assim, chamadas a funções implementadas externamente serão resolvidas por `script_lua_caller`, que as passará adiante para `script_call`.

No plugin Python, ao chamar uma função no módulo do ambiente virtual, o *callback* `__getattro` do módulo, definido como a função interna `script_python_get`, é chamado. Esta função procura uma entrada no dicionário do módulo e, caso não a encontre, cria um objeto do tipo `script_python_object`, e o retorna como resultado de `__getattro`. Este tipo de dado é declarado no plugin como uma classe Python, cujas instâncias contêm um ponteiro para o ambiente virtual e uma string C com o nome da função que eles representam. Estes objetos possuem o seu *callback* `__call` definido como `script_python_caller`, uma função que, assim como `script_lua_caller`, converte os parâmetros recebidos para o buffer de LibScript, invoca `script_call` e converte os valores de retorno de volta a Python. Assim, objetos deste tipo são *functors*, e se comportam de forma similar à *closure* definida no plugin Lua.

A resolução de funções sob demanda em Ruby é implementada utilizando o método `method_missing`, que é um fallback definido pela linguagem, chamado sempre que um método inexistente é invocado em uma classe. Diferentemente de `__getattro` em Python e `__index` em Lua, que são tratadores de acesso a atributos e portanto precisam retornar um objeto que é chamado em um passo seguinte, o método `method_missing` trata chamadas diretamente. Assim, ao ser invocado, `method_missing` recebe o nome do método solicitado e os parâmetros passados e os encaminha para a função `script_call`.

No plugin Perl, como em Lua e Python, também há uma função C responsável por realizar a a invocação de `script_call` e a conversão de parâmetros e valores de retorno. Esta função, `script_perl_caller`, para que possa ser exposta ao interpretador Perl, é implementada em um módulo XS. Uma vez carregado o módulo, a função é visível em Perl como a função `LibScript::caller`. A resolução dinâmica de funções do pacote Perl que representa o ambiente virtual LibScript é feita usando a função `AUTOLOAD` de Perl, que se comporta como `method_missing` em Ruby, capturando chamadas a funções inexistentes. Na função de inicialização do plugin, código Perl é executado para carregar o módulo de extensão, inicializar o pacote do ambiente e inserir nele uma função `AUTOLOAD` que chamará `LibScript::caller`:

```
snprintf(code, LEN_CODE,
   /* Inicializa o módulo de extensão */
   "bootstrap LibScript;"
   /* Declara o pacote do ambiente */
   "package %s;"
   /* Armazena o ponteiro do ambiente em Perl */
   "$LibScript::env = %p;"
   "sub AUTOLOAD {"
```

```
        "our $AUTOLOAD;"
        /* Extrai o nome do método
           do nome qualificado ``pacote::método'' */
        "$AUTOLOAD =~ s/[^:]*::/;"
        /* Invoca caller passando o endereço do ambiente, */
        /* o nome do método, e o array de argumentos */
        "LibScript::caller(%p, $AUTOLOAD, @_);"
    "}",
    state->package, env, env);
/* Avalia a string de código;
   TRUE indica que erros devem ser sinalizados. */
Perl_eval_pv(my_perl, code, TRUE);
```

4.2.5 Captura de erros

Os plugins devem capturar a ocorrência de erros na execução de strings de código e em chamadas de função. Em Lua, ambas as operações são realizadas usando a função `lua_pcall`, cujo valor de retorno indica a ocorrência de erros. No caso de erros, a mensagem de erro é obtida no topo da pilha virtual de Lua e propagada para a biblioteca principal usando `script_set_error_message`. No caso da execução de strings de código, erros de compilação são detectados através do valor de retorno da função `luaL_loadstring`, que carrega o código a ser executado por `lua_pcall`.

Em Python, a ocorrência de erros é sinalizada pelo valor de retorno das funções de execução de strings, `PyRun_SimpleString`, e de chamada de funções, `PyEval_CallObject`. No caso de erros, chamamos a função `PyErr_Occurred`, que retorna um objeto Python representando a exceção. A mensagem de erro é obtida convertendo este objeto para uma string Python usando `PyObject_Str`, e finalmente para uma string C com `PyString_AS_STRING`.

Em Perl, erros são sinalizados na variável especial `$@`, cujo conteúdo pode ser verificado através da API de C com a macro `ERRSV`. O teste para ocorrência de erros é `SvTRUE(ERRSV)`, e a mensagem de erro pode ser obtida convertendo esta variável para uma string C com a macro `SvPV`.

Ruby disponibiliza uma função para execução de strings de código, `rb_eval_string`, e uma versão desta que captura erros e sinaliza a sua ocorrência através do valor de retorno, `rb_eval_string_protect`. Entretanto, para chamadas de método, não há uma versão protegida da função `rb_funcall`. A função disponibilizada pela API para proteger chamadas, `rb_protect`, não recebe como parâmetro um método Ruby, mas sim uma função C. Para chamar métodos Ruby de forma protegida, precisamos escrever uma função C que encapsula a chamada:

```
static VALUE script_ruby_pcall(VALUE args) {
   /* Extrai nome do método do array de argumentos */
   ID fn_id = SYM2ID(rb_ary_pop(args));
   /* Extrai a classe do array de argumentos */
   VALUE klass = rb_ary_pop(args);
   return rb_apply(klass, fn_id, args);
}
```

e então invocá-la usando `rb_protect`:

```
/* Insere a classe no array de argumentos */
rb_ary_push(args, klass);
/* Insere o nome do método no array de argumentos */
rb_ary_push(args, ID2SYM(rb_intern(fn)));
/* Chama a função wrapper */
ret = rb_protect(script_ruby_pcall, args, &error);
if (error) {
   script_reset_buffer(env);
   script_set_error_message(env, StringValuePtr(ruby_errinfo));
   ruby_errinfo = Qnil;
   return SCRIPT_ERRLANGRUN;
}
```

Como a função `rb_protect` passa apenas um `VALUE` para a função C, precisamos armazenar a classe, o identificador do método e os parâmetros de entrada do método Ruby a ser invocado em um array Ruby. A ocorrência de erros é sinalizada em uma variável passada no terceiro parâmetro de `rb_protect`, e a mensagem de erro é obtida no `VALUE` global `ruby_errinfo`.

4.3 Conclusões

O estudo de caso apresentado aqui ilustrou, através da implementação dos plugins, o processo de embutir quatro linguagens de script realizando interface com uma mesma API C. Diversos aspectos da interação entre C e as linguagens de script foram abordados, contemplando inicialização e encerramento do ambiente de execução, passagem de dados e chamadas de função nos dois sentidos e a sinalização de erros. A partir disto, podemos fazer algumas observações sobre a adequabilidade destas linguagens como ambientes embutidos em aplicações.

Em muitas aplicações é importante que haja isolamento entre os scripts executados, como por exemplo, em scripts de diferentes clientes rodando em um servidor web. Como vimos, Lua e Perl permitem disparar múltiplos ambientes de execução, o que garante isolamento. Já Python e Ruby permitem apenas um estado, reduzindo sua aplicabilidade para cenários onde os scripts devem

executar isolados uns dos outros[6]. Estas duas linguagens trazem ainda outro problema: em alguns casos não é possível trazer o seu espaço de dados de volta ao estado original durante a execução de uma aplicação. Em Python, módulos importados não podem ser descarregados. Em Ruby, uma classe não pode ser redefinida (somente estendida) e `ID`s não são coletados.

Na implementação do plugin de Perl fica evidente que a sua API não foi projetada visando embutir o interpretador em aplicações. Além de exigir o desenvolvimento de um módulo de extensão para que o código Perl possa ter acesso a funções C, observamos aqui que a sua API é incompleta no que diz respeito ao seu uso como linguagem embutida. Muitas macros foram desenvolvidas assumindo que seriam sempre invocadas a partir de código escrito em arquivos XS, ou mesmo por código gerado pelo pré-processador XS. Isto é confirmado pela necessidade de passar parâmetros adicionais não-documentados para que as macros funcionem, como pôde ser observado na Seção 4.2.3.

Lua, por sua vez, mostrou-se apropriada como linguagem embutida, não compartilhando das limitações aqui descritas sobre as outras linguagens. Além disso, ela possui uma API simples, que trata as construções da linguagem de forma completa e ortogonal, o que se deve tanto ao foco da implementação de Lua como linguagem embutida, quanto ao projeto minimalista da linguagem em si. Mesmo em exemplos pequenos como os apresentados aqui, que exercitam apenas uma parte pequena das APIs, podemos observar que aspectos onde as linguagens definem tratamentos especiais ou possuem menor uniformidade transparecem nas APIs para C. Tanto em Python como em Ruby, funções que retornam múltiplos valores geram conversões implícitas para tipos agregados (listas e arrays). De forma similar, múltiplos retornos são representados em Perl através de contextos do tipo array. Nos seus respectivos plugins LibScript, estas características tiveram que ser tratadas de forma especial. No plugin Lua, em contraste, o tratamento para um valor único de retorno é igual ao de valores múltiplos, assim como ocorre na linguagem.

[6]Em Python é possível alternar a tabela de globais durante a execução de diferentes threads, o que oferece uma alternativa, um tanto mais trabalhosa, para obter isolamento. Ainda assim, o estado global compartilhado por módulos de extensão é o mesmo.