Estudo sobre APIs de linguagens de script

Hisham H. Muhammad

orientador: Roberto Ierusalimschy

Pontifícia Universidade Católica – Rio de Janeiro

28 de agosto de 2006

Interação entre linguagens

- Desenvolvimento multi-linguagem
  - Explorar os pontos fortes de cada uma em em diferentes partes de um sistema
- Diferentes modelos
  - Tradução de código
  - Compartilhamento de máquinas virtuais
  - Modelos de objetos independentes de linguagem
  - Interfaces de acesso externo

Linguagens de script

- Modelo de programação popular na atualidade, baseado no uso de duas linguagens
- Linguagens de programação de sistemas: compiladas, estaticamente tipadas
  - Implementação de componentes
- Linguagens de script: interpretadas, dinamicamente tipadas
  - Integração de componentes (*glue languages*)

APIs de linguagens de script

- Linguagens extensíveis
  - Estender a máquina virtual com novos recursos
  - Bindings de bibliotecas externas
  - Módulos de extensão
- Linguagens de extensão (*embedding*)
  - Estender uma aplicação através de scripts
  - Este segundo cenário engloba o primeiro

Objetivo

- Estudo sobre as principais problemas tratados no projeto de APIs de linguagens de script para C
- Linguagens abordadas: Python, Ruby, Perl, Lua, Java
- Duas partes:
  - Análise das APIs, comparando as soluções adotadas nas questões de projeto
  - Estudo de caso: implementação envolvendo as APIs de linguagens de script

Por que Java?

- Diferença principal
  - Tipagem estática
- Semelhanças
  - Máquina virtual
  - Coleta de lixo
  - Carga dinâmica de código
  - API para C

Questões no projeto de uma API

- Transferência de dados
  - Conversão de dados
- Gerência de memória
  - Coleta de lixo
- Chamada de funções
  - Passagem de parâmetros
  - Valores de retorno
- Registro de funções C

Transferência de dados

- Tratar diferença entre os sistemas de tipos
- Dados fluem entre linguagens de várias formas:
  - Parâmetros de entrada e valores de retorno
  - Atributos de objetos, estruturas de dados...
- Alternativas:
  - Expor o dado como tipo opaco (*handle*)
  - Converter o valor
  - Oferecer uma API para manipulação do dado

Abordagens na transferência de dados

- Conversão para tipos básicos
  - Funções para conversão de números, strings...
  - Java, devido à tipagem estática, realiza conversões automaticamente
- Tipos estruturados:
  - Via *handles*: Python (`PyObject`), Ruby (`VALUE`), Perl (SV), Java (`jobject`)
  - Lua permite manipulação de tabelas apenas através da API

Interação com dados de C

- Criação de dados contendo structs C
  - Fácil em Perl (SV contendo memória), Ruby (`Data_Wrap_Struct`) e Lua (*userdata*)
  - Complicado em Python
  - Não é possível em Java
- Armazenamento de ponteiros C
  - Fácil em Python (`PyCObject`), Lua (*light userdata*) e Perl (SV contendo ponteiro)
  - Em Java e Ruby, armazenam-se ponteiros como tipos numéricos (o que incorre em problemas de portabilidade)

Coleta de lixo

- Modelos diferentes de gerência de memória
  - C possui gerência explícita
  - Linguagens com coleta de lixo isolam o programador da gerência de memória
- Tempo de vida dos objetos
  - Dados de C referenciados na linguagem
  - Dados da linguagem referenciados em C

Diferentes graus de isolamento do coletor

- Perl e Python: contagem de referência, coleta de lixo explícita na API
- Ruby: abstrai a coleta na manipulação de objetos Ruby, exige implementação de função *mark* para uso do coletor *mark-and-sweep* em objetos contendo estruturas C
- Lua: abstrai a coleta com o modelo de API de pilha; coletor de lixo incremental aparece somente em `lua_gc`
- Java: abstrai totalmente o coletor; a única operação exposta é `System_gc`

Gerência de referências

- Lua e Ruby têm a gerência de referências mais transparente
  - Lua: no modelo de API de pilha o tempo de vida dos objetos é controlado por Lua
  - Ruby: varre a pilha de C para marcar variáveis locais (técnica não portável)
- Perl e Java definem 2 tipos de referências
  - Referências locais têm gerência implícita e só duram até o fim da função
- Mecanismos para valores globais
  - `rb_global_variable`, `luaL_ref/unref...`

Chamada de funções a partir de C

- A API deve prover uma forma para invocar em C funções da linguagem de script
- Devido à tipagem estática de C, não é possível usar uma sintaxe transparente para chamadas registradas em tempo de execução
  - Passar parâmetros
  - Especificar a função
  - Obter valores de retorno

APIs de chamada de funções

- Funções como objetos de primeira classe
  - Em Python, Lua e Perl funções são objetos
  - Em Ruby e Java, não -- tipos especiais em C para identificar funções
    - Java expõe implementação da JVM ao especificar funções
- Funções para realizar chamadas
  - Python, Java e Ruby oferecem grande número de funções de conveniência
  - Lua utiliza modelo simples de pilha
  - Perl expõe estrutura do interpretador através de um complexo protocolo de macros

Registro de funções C

- API para registro de funções C no espaço de dados da linguagem de script
- Em linguagens dinâmicas, funções registradas em tempo de execução podem ser acessadas transparentemente
- Em linguagens estáticas, elas devem ser declaradas a priori

Assinaturas de funções C

- Python e Ruby oferecem diferentes opções
- Lua oferece uma, apropriada para o modelo de pilha
- Em Java, as assinaturas são criadas com a ferramenta `javah`
  - Processa as classes Java e gera o cabeçalho C com os tipos de parâmetros adequados
- Em Perl, usando o pré-processador XS
  - A ferramenta encapsula as funções e gera as assinaturas e código de conversão de parâmetros.

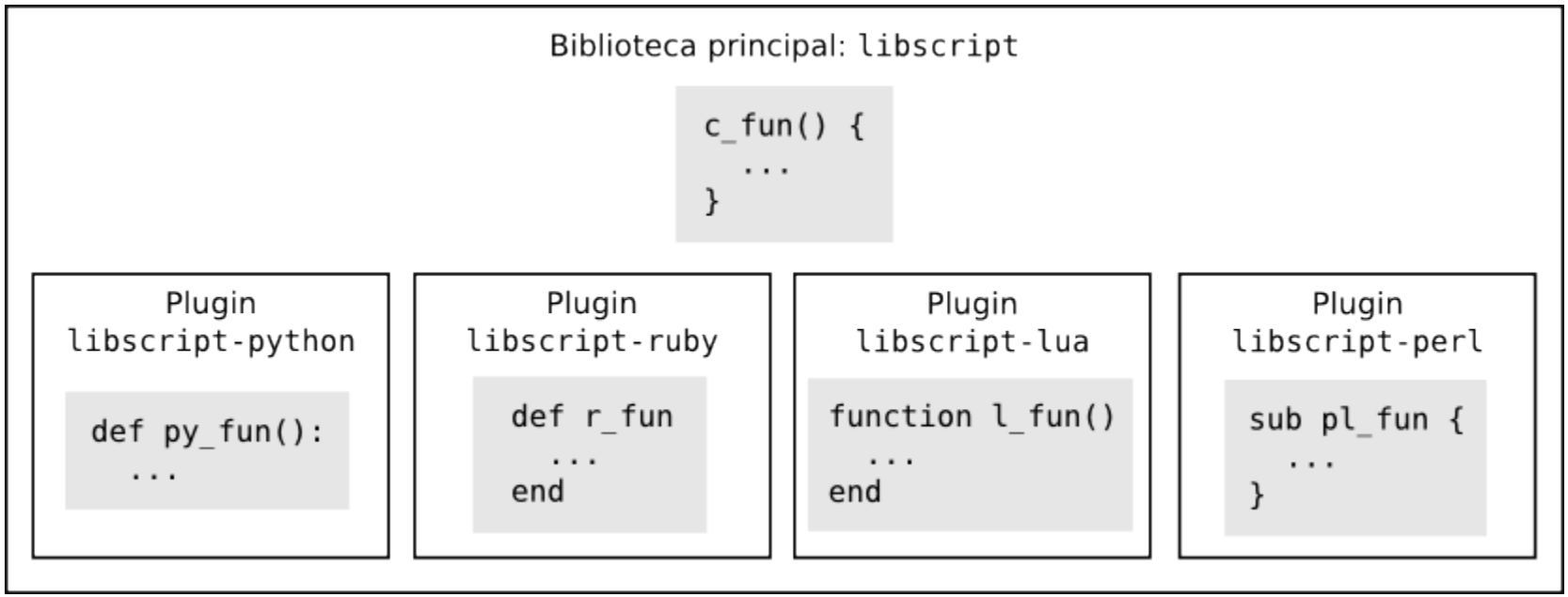
API para registro de funções

- Em Ruby e Lua é simples
  - Em Lua, é uma atribuição
- Python possui recursos para registro em lote
  - Não há todavia forma simples de registrar uma única função
- Em Java e Perl o registro é feito de forma implícita
  - Nenhuma das duas linguagens oferece API para registrar novas funções C durante a execução do programa

Estudo de caso: LibScript

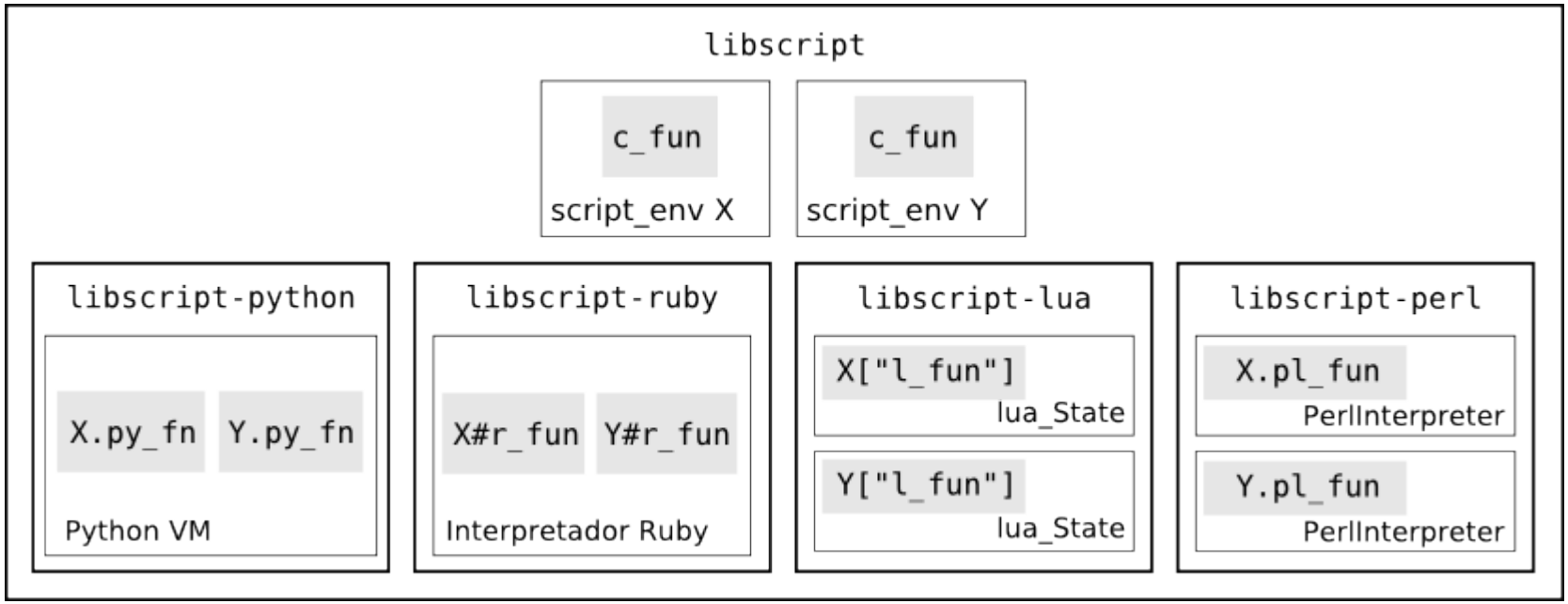
- Biblioteca projetada para tornar aplicações extensíveis através de scripting de forma independente de linguagem
- Baseada num modelo de plugins
- Comparar a implementação dos plugins nos permite comparar as diferentes APIs realizando tarefas equivalentes
- Exercita as linguagens como linguagens de extensão (*embedding*)

LibScript: visão geral



- Funções C registradas na biblioteca principal
- Plugins carregados dinamicamente

Ambientes virtuais

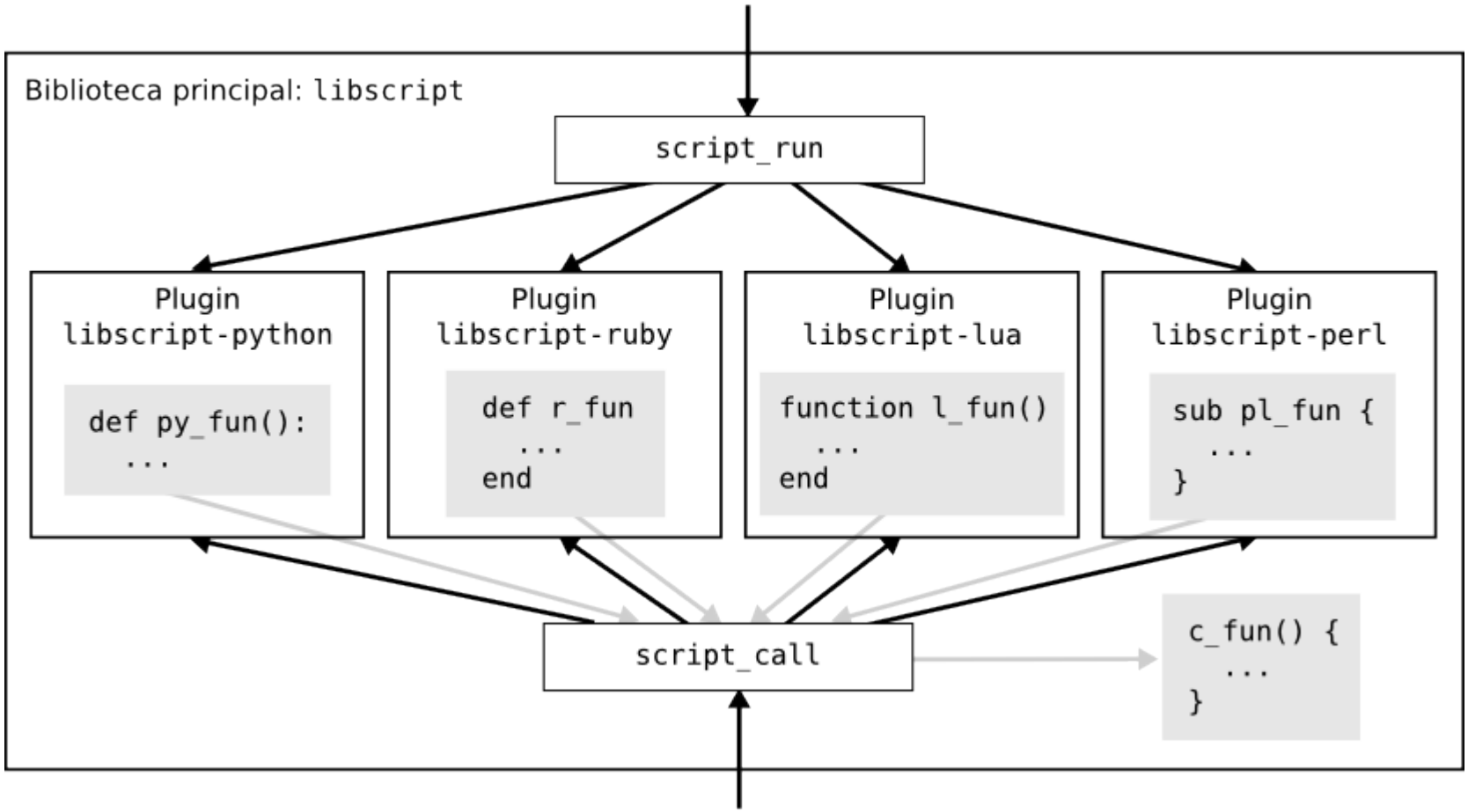


- Um ambiente virtual define um espaço de nome
- Ambientes são isolados quando possível
- Cada linguagem possui sua representação do ambiente virtual

API de LibScript

- Inicialização e término
  - script_init, script_done
- Registro de funções
  - script_new_function
- Buffer de parâmetros
  - script_{get,put}_{double,int,bool,string}
- Execução de código
  - script_run{,_file}, script_call
- Tratamento de erro
  - script_error, script_{,set_}error_message

Execução de código



API de plugins

- Plugins devem implementar quatro operações:
  - Init: responsável por inicializar o plugin, disparar uma instância da máquina virtual, criar o ambiente virtual
  - Run: recebe uma string de código a ser executado
  - Call: chama funções declaradas na representação local do ambiente virtual
  - Done: encerra o ambiente virtual

Implementação dos plugins

- Isolamento dos ambientes de execução
  - Possível somente em Perl e Lua
  - Python e Ruby, ainda, não permitem limpar o estado de dados de volta ao estado original
- API de Perl se mostrou inadequada para *embedding*
  - Parte do código teve que ser implementado como um módulo de extensão usando XS
- Tratamento de valores de retorno
  - Casos especiais em Python, Ruby e Perl

Conclusões

- Aplicações embutindo uma máquina virtual demandam mais da API do que a implementação de módulos de extensão
  - Dificuldade no registro e acesso a globais em Python
  - Complexidade da API de chamadas em Perl
- A tipagem estática de Java traz ganhos limitados na interação com C
  - A ligação de campos e métodos ocorre de forma dinâmica, com comportamento diferente de como ocorre em código Java

Conclusões

- Documentação
  - Python, Lua e Java: APIs bem documentadas
  - Ruby, Perl: documentação incompleta
- Papel importante da disponibilidade da documentação na *definição* da API
  - Python: 656 funções públicas
  - Lua: 113 (79 *core*, 34 biblioteca auxiliar)
  - Java: 228
  - Ruby: 530, Perl: 1209 (...mas quantas fazem parte da API?)

Conclusões

- Equilíbrio entre simplicidade e conveniência
  - Python: API extensa, funções que abreviam seqüências de chamadas
  - Lua: API minimalista, oferece mecanismos para compor funcionalidade
- Consistência da API depende da consistência da linguagem
  - Problemas no tratamento de blocos em Ruby
  - Contextos de execução de funções em Perl

Conclusões

- Traçamos um panorama dos problemas gerais enfretados na interação entre C e o ambiente de execução de linguagens de script
- Apresentamos como as APIs de cinco linguagens tratam estes problemas
- Realizamos uma comparação prática do uso de APIs de linguagens de script através do estudo de caso

Possibilidades de trabalhos futuros

- Este trabalho realizou uma análise qualitativa
  - Outros aspectos que afetam o projeto de APIs podem ser considerados, como desempenho
- Continuidade do desenvolvimento de LibScript
  - Revisão da API
  - Novos plugins
  - Aplicações
- http://libscript.sourceforge.net