2
Interação entre linguagens de programação

As abordagens aplicadas na interação entre diferentes linguagens de programação variam bastante entre si, mas é possível identificar algumas técnicas tipicamente utilizadas: tradução de linguagens, seja de uma para outra ou de ambas para uma terceira; comunicação através de protocolo ou linguagem intermediária; compartilhamento de um ambiente de execução comum, seja de máquina virtual ou através de convenção de chamadas; e interfaces de acesso externo.

2.1
Tradução de código

Permitir o uso de duas linguagens diferentes em um programa traduzindo o código de uma delas para a outra minimiza o problema de comunicação entre as partes do programa escritas em linguagens diferentes, uma vez que o programa final utilizará um espaço único de dados. Em contrapartida, ao ter de se descrever uma linguagem em termos de outra, a diferença de semântica das construções pode se tornar um problema. Se a linguagem de destino não possui construções oferecidas pela linguagem de origem, simulá-las pode ser custoso.

Um exemplo típico de problema enfrentado em ferramentas de tradução de código é a complexidade adicionada pela simulação de funções de alta ordem e recursão final ao traduzir-se o código de linguagens funcionais para outra que não possui estes recursos. Tarditi et al. (Tarditi 1992) descrevem o desenvolvimento de um tradutor de Standard ML para ANSI C. Medições evidenciaram o custo de adaptação destes recursos de ML para C, resultando em código em média 2 vezes mais lento que o gerado pelo compilador ML nativo. Em (Tolmach 1998), são discutidos desafios similares na tradução de ML para Ada: na solução adotada, o processo possui um passo intermediário onde as construções de alta ordem são "aplainadas" para construções de primeira ordem envolvendo registros, para que pudessem assim ser representados em Ada.

Além de problemas como este, diferenças na representação dos dados

PUC-Rio - Certificação Digital Nº 0410837/CB

ainda é algo a ser tratado no processo de tradução de uma linguagem para outra. No caso particular de C, as facilidades de baixo nível para manipulação de memória permitem a descrição de estruturas de dados de linguagens de mais alto nível sem maiores problemas. Isto faz de C uma candidata freqüente para uso como representação de baixo nível portável. O compilador de Haskell GHC oferece, como alternativa à geração de código nativo, geração de código C para uso com o GCC (Jones 1993). Uma das vantagens deste recurso é permitir o *bootstrapping* do compilador em novas arquiteturas, dado que o próprio GHC é escrito em Haskell. De fato, a ubiqüitude dos compiladores C faz com que a linguagem seja utilizada também como *lingua franca* entre diferentes linguagens, como será visto na Seção 2.4.

2.2 Compartilhamento de máquinas virtuais

Outra abordagem para a interação entre linguagens envolve o uso de um ambiente de execução comum, como uma máquina virtual. O código das diferentes linguagens é compilado de modo a produzir representações compatíveis entre si, de acordo com os tipos de dados oferecidos pelo ambiente de execução. Diversas implementações utilizam a Java Virtual Machine (Lindholm 1999) com essa finalidade. Jython (Hugunin 1997) é uma implementação da linguagem Python que produz bytecodes Java. SMLj (Benton 1999) é um compilador Standard ML que gera bytecodes Java e permite acesso de classes e métodos Java a estruturas e funções ML e vice-versa. O fato da Java Virtual Machine não ter sido projetada para comportar diferentes linguagens de programação, entretanto, transparece nas limitações apresentadas por estes projetos. SMLj define extensões à linguagem ML para permitir acesso a construções específicas de Java; Jython possui limitações ao realizar a interface de Python com a API de reflexão e carga dinâmica de classes de Java. Além disso, o conjunto de instruções da máquina virtual privilegia operações que condizem com a semântica de Java, tornando, por exemplo, implementações de *arrays* com semântica diferente menos eficientes.

O .NET Framework (Box 2002) é um ambiente de execução baseado em máquina virtual que vem sendo indicado pela Microsoft como plataforma de programação preferencial em sistemas Windows. Apesar de com ele ter sido introduzida a linguagem C# (ISO 2006), o ambiente tem como um de seus objetivos oferecer suporte a múltiplas linguagens – evidenciado pelo próprio nome Common Language Runtime (CLR) – em contraste com as limitações que o ambiente de Java impõe àqueles que tentam utilizá-lo com outras linguagens. Todavia, adaptações às linguagens continuam necessárias

PUC-Rio - Certificação Digital Nº 0410837/CB

com o ambiente .NET: A versão .NET de Visual Basic inclui alterações na linguagem de modo a tornar a sua semântica mais similar à de C#; um novo dialeto de C++, C++/CLI, foi introduzido adaptando o modelo de gerenciamento de memória ao do CLR (ECMA 2005); de forma análoga, foi desenvolvido um novo dialeto de ML chamado F#, para, entre outros motivos, proporcionar melhor integração com componentes .NET desenvolvidos em outras linguagens (Syme 2006).

Outra implementação de uma máquina virtual para múltiplas linguagens vem sendo feita pelo projeto Parrot (Randal 2004). O escopo deste projeto é mais restrito, visando ser um *back-end* comum para linguagens dinâmicas como Perl e Python. O foco do projeto, entretanto, está atualmente na implementação de Perl 6.

Um tipo de comunicação que pode ser considerado também uma forma de ambiente de execução comum é a comunicação entre executáveis e bibliotecas nativas através de convenções de chamada: regras para a passagem de parâmetros na pilha de execução, uso de registradores e *mangling* de nomes. Este pode ser considerado o método de interação entre código em diferentes linguagens de mais baixo nível. Convenções de chamada, entretanto, são um recurso limitado de comunicação, já que assumem tipos de dados com representação em memória idêntica nas duas linguagens. Tal compatibilidade dificilmente ocorre a não ser que uma das linguagens explicitamente considere este tipo de interação na sua definição: o padrão de Ada, por exemplo, requer que as suas implementações sejam compatíveis com as convenções de C, COBOL e Fortran (Ada 1995). De forma similar, C++ permite especificar funções com linkagem compatível com C, através da diretiva `extern "C"`.

PUC-Rio - Certificação Digital Nº 0410837/CB

2.3 Modelos de objetos independentes de linguagem

Adotar um modelo de tipos independente de linguagem é uma outra forma de tratar as questões de interoperabilidade de dados entre linguagens. Assim, na definição dos dados de uma aplicação, as suas interfaces são definidas de forma neutra, tipicamente utilizando alguma linguagem projetada especificamente para este fim (uma IDL, *interface description language*) enquanto as implementações são feitas nas linguagens específicas. A arquitetura CORBA (*Common Object Request Broker Architecture*) (OMG 2002) é uma das principais representantes deste modelo. A motivação principal para o desenvolvimento de CORBA foi permitir o desenvolvimento de aplicações distribuídas em ambientes heterogêneos; a heterogeneidade de linguagens foi um dos aspectos levados em consideração.

Os desafios existentes ao projetar um modelo de dados ou objetos "independente de linguagens", entretanto, são parecidos com os de uma interface entre duas linguagens quaisquer, já que esse modelo, por sua vez, descreve também um sistema de tipos. Ao implementar *bindings* para algum destes modelos de objetos é necessário definir uma correspondência entre os tipos definidos pelo modelo e os oferecidos pela linguagem destino e prover a esta uma API para interação com o ambiente de execução – no caso de CORBA, com o ORB (*Object Request Broker*).

Se por um lado a tarefa pode ser facilitada pelo fato de o modelo ter sido projetado visando interação com outras linguagens (diferentemente, por exemplo, do sistema de tipos de C), por outro espera-se usualmente um grau de transparência maior na representação dos dados. Por exemplo, enquanto em uma aplicação integrando C++ e Python a distinção entre objetos C++ e objetos Python é clara e a API Python define o limite entre os dois universos, em uma aplicação desenvolvida utilizando CORBA espera-se que, tanto em uma linguagem como em outra, a manipulação dos objetos seja igual, sejam eles implementados em C++ ou Python. Para isso, a solução adotada é o uso de *stubs*, objetos que dão uma aparência nativa uniforme aos dados, indepentemente da linguagem em que foram implementados e, no caso de modelos distribuídos como CORBA, da localização dos mesmos na rede. A correspondência entre os ciclos de vida dos *stubs* e dos objetos que eles representam é outro fator que deve ser levado em consideração. Nos *bindings* Java, por exemplo, isto é realizado com o auxílio do coletor de lixo da própria linguagem. Já em linguagens como C++ o controle das referências é explícito.

Outras abordagens de mais alto nível têm sido propostas para a integração de aplicações desenvolvidas em múltiplas linguagens. Linguagens de coordenação como Linda (Gelernter 1985) e Opus (Chapman 1997) definem mecanismos para troca de mensagens e um conjunto restrito de construções para indicar o fluxo destas entre agentes implementados em outras linguagens.

2.4 C como linguagem intermediária

O desejo de uma linguagem intermediária universal é antigo no mundo da computação. Diversas propostas surgiram ao longo dos anos, desde o projeto UNCOL (Conway 1958) às linguagens de sintaxe extensível da década de 70 (Metzner 1979) até os mais recentes ambientes de máquina virtual como .NET. Na prática, as necessidades que estes projetos visavam atender vêm sendo supridas ao longo dos anos de forma mais pragmática, ainda que menos ideal, usando C. Dois motivos fazem de C uma escolha comum como linguagem

PUC-Rio - Certificação Digital Nº 0410837/CB

intermediária. Primeiro, a sua característica de linguagem de "médio nível", ao permitir ao mesmo tempo independência de hardware e manipulação direta de memória. Segundo, a grande disponibilidade de compiladores C, alavancada pela proliferação dos sistemas Unix nas mais diferentes arquiteturas. Assim, com o passar do tempo, oferecer uma interface para interoperabilidade com outras linguagens passou a significar oferecer uma interface para comunicação com código C. Isto é especialmente verdadeiro para linguagens dinâmicas que oferecem recursos para extensibilidade de aplicações. Não por acaso, estas são tipicamente implementadas em C.

A disponibilidade de APIs para C oferecidas pelas diferentes linguagens faz com que C seja bastante usada também como "ponte". A integração entre Python e Fortran se dá através de um módulo Python escrito em C que acessa uma biblioteca Fortran, que por sua vez expõe funções com convenção de chamada compatível com C (Peterson 2001). LunaticPython (Niemeyer 2006) oferece pontes de Lua para Python e de Python para Lua, implementadas através de um par de módulos de extensão para cada linguagem de origem escritos em C.

Todavia, linguagens intermediárias genéricas continuam a ser propostas como alternativas a C. C-- (Jones 1999) é um projeto que tenta superar as limitações de C enquanto linguagem intermediária tornando mais explícita a representação em memória dos tipos de dados e adicionando suporte a construções não facilmente representáveis em C, como recursão final. Versões recentes da suíte de compiladores GCC padronizaram uma linguagem intermediária para comunicação entre os seus diversos *back-ends* e *front-ends* (Dvorak 2005).

2.5 Interfaces com C

A linguagem C tem, na atualidade, um papel especial no mundo das linguagens de programação. Além de ser bastante utilizada na implementação de compiladores, interpretadores e máquinas virtuais (as principais implementações de Perl, Python, Ruby, e Lua são apenas alguns exemplos), é também usada em compiladores como formato de saída na geração de código portável (dois exemplos notáveis são os compiladores GHC e SmartEiffel (Collin 1997), que geram C a partir de Haskell e Eiffel, respectivamente). Isto faz com que a API para C seja um formato conveniente para uma interface de acesso externo.

Na grande maioria dos casos, a representação interna de código produzida por compiladores de outras linguagens não é compatível com C, seja por diferenças em convenções de chamadas ou de nomes, ou por produzirem

PUC-Rio - Certificação Digital Nº 0410837/CB

código para execução em máquinas virtuais. Desta forma, para permitir a um programa em C acessar este código, cabe à linguagem expor uma biblioteca de funções C que realizem a tradução necessária. Em ambientes de máquina virtual, esta biblioteca de funções normalmente é genérica, oferecendo facilidades para comunicação com a própria máquina virtual. Para linguagens estáticas, usualmente é necessário criar uma biblioteca específica que realize a conversão das chamadas, como ocorre em interfaces que expõem bibliotecas C++ para C. Um exemplo disto é QtC (KDE 2006), uma biblioteca de bindings C para o toolkit gráfico Qt, que é implementado em C++.

Para linguagens não imperativas, há ainda o problema de código C potencialmente gerar efeitos colaterais. Algum recurso para isolamento das chamadas deve ser oferecido. No GHC, a construção para chamadas em C, `_ccall_`, é definida na mônada de IO; no adendo do padrão Haskell 98, a diretiva `ccall` foi integrada, mas o uso da mônada é opcional, cabendo ao programador garantir que funções que não a utilizem sejam puras[1].

Outra possível fonte de incompatibilidade entre linguagens que deve ser tratada quando estas interagem é a diferença entre modelos de concorrência. C, em particular, não define construções de concorrência, sendo estas implementadas através de bibliotecas. Ao mesmo tempo que isto traz grande flexibilidade à linguagem, implica também em problemas de portabilidade para linguagens que dependam da disponibilidade de mecanismos de concorrência em C compatíveis com os modelos que elas utilizam.

Por exemplo, APIs entre C e Java devem levar em conta o modelo de *multithreading* preemptivo adotado por Java. A JNI (*Java Native Interface*) (Liang 1999) define funções para controlar exclusão mútua entre dados compartilhados entre as duas linguagens. O programador deve tomar o cuidado de buscar o equilíbrio entre o tempo gasto bloqueando a máquina virtual acessando dados compartilhados e o tempo gasto realizando cópia de dados entre os ambientes para reduzir o compartilhamento. Outra situação em que o modelo de concorrência da linguagem demanda cuidados por parte do programador na integração com C ocorre no uso de co-rotinas em Lua. A combinação de dois recursos de Lua, multitarefa cooperativa com múltiplas pilhas de execução e a capacidade de alternar entre chamadas a funções Lua e C em uma pilha, traz consigo uma limitação: uma co-rotina não pode executar a operação de *yield* caso haja uma função C na sua pilha, uma vez que não há uma forma portável de alternar entre múltiplas pilhas em C (Moura 2004).

Uma das motivações mais freqüentes para integração com código C é

[1]Diversas convenções de chamada são definidas (`stdcall`, `cplusplus`, `jvm`, `dotnet`) mas `ccall` é a única declarada como obrigatória pelo documento.

PUC-Rio - Certificação Digital Nº 0410837/CB

o uso de bibliotecas externas. Expor uma biblioteca C através da FLI para acesso em uma linguagem pode incorrer no registro de centenas de funções. É comum definir também tipos de dados que dêem às estruturas definidas pela biblioteca uma aparência mais nativa, como por exemplo converter funções C que registram *callbacks* em métodos Ruby que aceitam blocos de código como parâmetro. Essas inicializações e adaptações são usualmente definidas em uma biblioteca de *bindings*, que serve de ponte entre a linguagem e a biblioteca C encapsulando a interação com a FLI.

Os padrões que ocorrem na produção de bindings são tão comuns que deram origem a programas que visam automatizar o processo. Estes geradores de bindings costumam trabalhar a partir de alguma representação preparada para o seu uso, já que analisar os cabeçalhos C pode-se mostrar insuficiente: por exemplo, o programa muitas vezes não seria capaz de interpretar a intenção de uma construção como `int**`. SWIG (Beazley 1996) é uma ferramenta multi-linguagem para geração de bindings de bibliotecas C e C++ popular que utiliza um formato próprio para descrição de interfaces. FLIs podem ainda utilizar geradores de *stubs* para poupar o programador de escrever código C repetitivo ou não portável. Java possui um gerador de cabeçalhos C contendo os protótipos dos métodos nativos a serem implementados. Pyrex (Ewing 2006) é um gerador de módulos C para Python a partir de uma sintaxe baseada na própria linguagem Python. Outro exemplo é toLua++ (Manzur 2006), uma ferramenta para integrar código C e C++ a Lua, que gera stubs a partir de arquivos de cabeçalhos C preparados para uso pelo programa, podendo conter anotações que auxiliem o processo de conversão.

PUC-Rio - Certificação Digital Nº 0410837/CB

2.6 Linguagens de script

Um modelo de interação entre linguagens que tem se mostrado especialmente relevante na atualidade é o que se dá entre linguagens compiladas tipadas estaticamente, como C e C++, e linguagens interpretadas tipadas dinamicamente, como Perl e Python. Em (Ousterhout 1998), Ousterhout categoriza estes dois grupos como *linguagens de programação de sistemas* e *linguagens de script*.

Estas duas categorias de linguagens possuem objetivos fundamentalmente diferentes. Linguagens de programação de sistemas surgiram como alternativa ao assembly no desenvolvimento de aplicações, tendo como principais características a tipagem estática, o que facilita a compreensão das estruturas de dados de sistemas grandes, e a implementação através de compiladores, devido a preocupações com o desempenho. Em contraste, linguagens de

script são tipadas dinamicamente e implementadas como interpretadores ou máquinas virtuais. A tipagem dinâmica e o uso extensivo de construções de alto nível como tipos básicos, como listas e hashes, traz maior flexibilidade na interação entre componentes; em linguagens estáticas, o sistema de tipos impõe restrições a estas interações, muitas vezes exigindo que o programador escreva interfaces de adaptação, o que torna o reuso de componentes mais difícil.

Ousterhout aponta que, num modelo integrando estes dois tipos de linguagens, a tendência é que linguagens de programação de sistemas não sejam mais usadas para escrever aplicações inteiras, mas passem a ser usadas para a implementação de componentes, que por sua vez são conectados através de código escrito com linguagens de script. A conveniência oferecida por linguagens interpretadas de alto nível permite prototipagem rápida e encoraja o reuso de componentes.

2.6.1 Linguagens extensíveis e de extensão

Com a popularização de linguagens de script, o modelo de desenvolvimento baseado em duas linguagens, passou a ser encarado de duas formas: além do modelo onde a linguagem de script tem função acessória, onde scripts escritos pelo usuário permitem customizações de aplicações, há também um modelo onde a linguagem de script desempenha um papel mais central na execução. Exemplos típicos são aplicações gráficas onde a interface é descrita por linguagens de script controlando componentes implementados em C e jogos onde a lógica é descrita em scripts e a *engine* de execução é implementada em linguagens de mais baixo nível.

Nestes cenários, existe claramente uma distinção entre uma camada de mais baixo nível onde desempenho é um fator crítico e outra de alto nível que se caracteriza por operações de coordenação de elementos da camada inferior. Linguagens de script deixam de ser apenas um mecanismo de extensão: a aplicação em si é escrita usando a linguagem de script e bibliotecas escritas em linguagens de mais baixo nível são carregadas por esta como módulos de extensão.

Faz sentido então, ao discutirmos interação entre linguagens, fazermos a distinção entre *linguagens extensíveis* e *linguagens de extensão*. Linguagens extensíveis são aquelas que podem ser estendidas através de módulos externos implementados em outras linguagens. Linguagens de extensão são aquelas cujo ambiente de execução pode ser embutido em uma aplicação, permitindo assim usá-las para estender a aplicação. Tipicamente, linguagens de script podem ser usadas, com maior ou menor grau de conveniência, como linguagens extensíveis

PUC-Rio - Certificação Digital Nº 0410837/CB

e linguagens de extensão. O foco da linguagem em um ou outro cenário tende a se refletir no projeto de sua FLI, como veremos na comparação entre as APIs de diferentes linguagens ao longo do Capítulo 3.

Outra observação interessante é que, enquanto em um modelo a linguagem de script serve como linguagem de extensão para a linguagem de baixo nível em que a aplicação é escrita, no outro modelo ocorre o contrário: podemos encarar módulos de extensão escritos utilizando a FLI como uma forma de estender a linguagem de script usando C como linguagem de extensão. Desta forma, o conjunto de funcionalidades providas por uma API entre C e uma linguagem de script tende a ser simétrico nos dois sentidos caso se queira permitir tanto extensibilidade da linguagem como o seu uso como linguagem de extensão.

2.6.2 Tcl

A integração de programas desenvolvidos em linguagens de programação de sistemas coordenados através de linguagens de script é prática comum há bastante tempo. Shell scripting em sistemas Unix é provavelmente o exemplo mais célebre, onde construções como pipes (que conectam a saída de um processo à entrada de outro) permitem realizar tarefas combinando uma série de programas implementados em outras linguagens, ou mesmo outros scripts.

Aplicações gráficas interativas não se encaixam bem a este modelo de programabilidade. A partir desta observação, Ousterhout identificou a necessidade de mover a linguagem de coordenação para *dentro* das aplicações, criando a linguagem Tcl (Ousterhout 1994), que adaptou o modelo de interação promovido por linguagens de shell para a manipulação de funções de uma aplicação.

No modelo de Tcl, a linguagem de script é implementada como uma biblioteca e é embutida em uma aplicação escrita em uma linguagem de mais baixo nível, como C. Estruturas de dados da aplicação são expostas ao ambiente de script como objetos manipuláveis; em contrapartida, a aplicação pode disparar funções na linguagem de script e acessar os seus dados.

Quatro objetivos principais foram destacados no projeto original de Tcl (Ousterhout 1990): foco como linguagem para comandos (voltada para escrever programas curtos); extensibilidade; simplicidade na implementação; e interface simples com aplicações C. Desde já observamos os princípios que hoje são entendidos como os fundamentos de linguagens extensíveis e de extensão: extensibilidade foi listada como objetivo explicitamente; os dois últimos objetivos (e o último em particular) demonstram o foco de Tcl como

PUC-Rio - Certificação Digital Nº 0410837/CB

linguagem de extensão.

Tendo em vista simplificar a interação com código C, Tcl utiliza strings como o tipo único de dados (conforme o contexto, strings podem ser interpretadas como por exemplo listas, números ou blocos de código Tcl). A API da linguagem, então, realiza a comunicação entre código C e o interpretador Tcl através de strings C. A influência do modelo de interação entre programas em shell scripting transparece na forma como código C se comunica com funções Tcl: parâmetros são recebidos em C a partir de Tcl usando um inteiro e um array C de strings, como nos parâmetros da função `main` de C. Uma função C a ser exposta para o interpretador Tcl deve seguir a seguinte assinatura:

```
typedef int (*Tcl_CmdProc) (ClientData c, Tcl_Interp *i,
                            int argc, char* argv[]);
```

Chamadas a Tcl a partir de C são feitas enviando uma string de código Tcl para execução:

```
int Tcl_Eval(Tcl_Interp *i, char* cmd);
```

O valor de retorno indica o sucesso ou erro da função, e a string de retorno do código Tcl pode ser obtida em `i->result`. Embora simples, esta abordagem possui algumas limitações significativas: a conversão constante de strings afeta o desempenho negativamente; além disso, a API limitada a strings faz com que código C tenha que realizar uma série de conversões explicitamente, mesmo em operações tão básicas quanto passar um inteiro como parâmetro.

Com o passar dos anos, outras linguagens de script apresentaram propostas diferentes para interação com código C. Além disso, o minimalismo de Tcl, que se mostrou uma vantagem da linguagem enquanto linguagem de extensão, faz com que ela pareça limitada frente a linguagens como Python, que provêem uma base mais completa enquanto linguagem extensível. As linguagens de script foram além do primeiro objetivo de Tcl (foco como linguagem de comandos, para programas curtos). Assim, Tcl gradualmente perdeu espaço como linguagem de script. Neste trabalho, nosso foco irá se concentrar em APIs modernas de linguagens de script populares na atualidade.

De fato, versões mais recentes de Tcl adotam um modelo de dados similar ao de linguagens de script mais modernas (Welch 1995), abandonando a manipulação puramente por strings ao definir um tipo C `Tcl_Obj` que corresponde a um valor Tcl. Esta nova API, assim, não é muito diferente da API de Python, que também possui um tipo C `PyObject`, que será discutido na Seção 3.1.1. Embora estas mudanças resolvam alguns dos problemas de desempenho da versão original de Tcl, elas trazem também outros problemas:

PUC-Rio - Certificação Digital Nº 0410837/CB

por questões de compatibilidade, objetos carregam ainda a representação do dado como string; além disso, contagem de referências passa a ser uma preocupação (também compartilhada pela API de Python – o problema é discutido na Seção 3.2.1).

A importância histórica de Tcl, entretanto, é inegável. Embora aplicações programáveis já existissem bem antes do seu surgimento, tipicamente usando pequenas linguagens criadas especificamente para a aplicação, foi o conceito introduzido por Tcl de implementar linguagens de script como bibliotecas para C que impulsionou fortemente o desenvolvimento de aplicações extensíveis.

PUC-Rio - Certificação Digital Nº 0410837/CB