5
Conclusão

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

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

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

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

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

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

Conclusões obtidas durante a implementação da biblioteca LuaInterface:

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

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

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

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

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

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

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

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

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

Facilidade de implementação é uma vantagem importante, quando

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

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

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

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

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

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