PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Um Compilador de Lua VM para LLVM

Gabriel de Quadros Ligneul

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC
DEPARTAMENTO DE INFORMÁTICA
Curso de Graduação em Ciência da Computação

Rio de Janeiro, Junho de 2016

Gabriel de Quadros Ligneul

Um Compilador de Lua VM para LLVM

Relatório de Projeto Final, apresentado ao Departamento de Informática da PUC-Rio como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Roberto Ierusalimschy

Rio de Janeiro
Junho de 2016

Agradecimentos

Gostaria de agradecer, sem ordem de prioridade:
Ao meu orientador Roberto, por toda a passagem de conhecimento.
À PUC e seus professores, pela inigualável formação.
Ao Tecgraf, pelo oportunidade de trabalho e apoio financeiro.
Aos colegas da PUC, por enfrentar essa jornada comigo.
Aos amigos do Tecgraf, por me ensinar a trabalhar em grupo.
À minha família, por todo o apoio dado durante minha graduação.
Ao meu avô Fernando, por dividir a casa próxima à faculdade.
À minha amiga Thamiris, pela companhia nos últimos anos.

Resumo

Ligneul, Gabriel de Quadros; Ierusalimschy, Roberto. **Um Compilador de Lua VM para LLVM**. Rio de Janeiro, 2016. 51p. Relatório de Projeto Final — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Lua é uma linguagem de extensão que possui diversas características de alto nível. Entre essas características, a interpretação é a que permite o código fonte ser executado sem uma compilação prévia. Contudo, a interpretação possui um grande custo computacional. Então, com intuito de melhorar o desempenho do interpretador de Lua, investimos na técnica de compilação *just-in-time*. Nessa técnica, o programa fonte é compilado em código de máquina durante a execução do interpretador. Para facilitar a geração do código de máquina, utilizamos a cadeia de ferramentas LLVM. A LLVM utiliza uma representação intermediária que permite a geração de código de máquina otimizado de forma portátil.

Palavras-Chaves

Linguagens de Programação; Compiladores; JIT; Lua.

Abstract

Ligneul, Gabriel de Quadros; Ierusalimschy, Roberto. **A Compiler from Lua VM to LLVM**. Rio de Janeiro, 2016. 51p. Relatório de Projeto Final — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Lua is an extension language which has several high level features. Among those features, the interpretation is the one that allows the source code to be executed without a previous compilation. However, the interpretation has a huge computational cost. So, with the intention to improve the performance of the Lua interpreter, we have invested in the just-in-time compilation technique. In this technique, the source program is compiled into machine code during the interpreter execution. In order to ease the machine code generation, we have used the LLVM toolchain. The LLVM uses an intermediate representation that allows the generation of optimized machine code in a portable way.

Keywords

Programming Languages; Compilers; JIT; Lua.

Sumário

1 Introdução **8**
- 1.1 Definição do Problema . . . 8
- 1.2 Soluções Existentes . . . 9
- 1.3 Proposta e Objetivos do Projeto . . . 9
- 1.4 Estrutura do Relatório . . . 10

2 Cadeia de Ferramentas LLVM **12**
- 2.1 Representação Intermediária e Forma SSA . . . 12
- 2.2 Interface com C e C++ . . . 17
- 2.3 Modificação do Compilador de Monga . . . 18
  - 2.3.1 Compilador Original de Monga . . . 18
  - 2.3.2 Modificações Feitas . . . 19
  - 2.3.3 Resultados Obtidos . . . 19

3 Componentes Internos do Interpretador de Lua **22**
- 3.1 Representação de Valores . . . 22
- 3.2 Pilha de Registradores . . . 23
- 3.3 Funções . . . 25
- 3.4 Instruções Portáteis . . . 26
- 3.5 Máquina Virtual . . . 29

4 Compilador de Lua bytecode para LLVM IR **31**
- 4.1 Interação com Interpretador de Lua . . . 32
- 4.2 Compilação das Instruções Portáteis de Lua . . . 33
- 4.3 Representação de Valores Lua Durante a Compilação . . . 34
- 4.4 Testes e Depuração . . . 37
- 4.5 Interface com o Usuário . . . 38

5 Testes de Desempenho **40**
- 5.1 Especificação dos Testes . . . 40
- 5.2 Avaliação do Tempo de Execução . . . 41
- 5.3 Avaliação do Tempo de Compilação . . . 43
- 5.4 Comparações com LuaJIT . . . 44

6 Considerações Finais **46**

A Exemplo de uso da interface C++ da LLVM **48**

Referências **50**

Lista de Figuras

2.1 Etapas de um compilador de C para x86 que usa LLVM. 12
2.2 Acima, uma função simples em C e, abaixo, a função IR gerada pelo compilador Clang. 13
2.3 Acima, uma função em C com uma condicional. Abaixo, uma função IR equivalante que exemplifica o uso de blocos básicos. Escrevemos essa função em IR manualmente ao invés de utilizar o compilador Clang. 15
2.4 Acima, uma função em C com um laço *for*. Abaixo, a função em IR equivalente. 16
2.5 Função fatorial em Monga. 18
2.6 IR gerado pelo compilador de Monga para a função fatorial da figura 2.5. 20
2.7 Comparações dos tempos de execução em relação aos executáveis gerados por Monga. 21

3.1 Exemplos de valores em Lua. 23
3.2 Exemplo das etapas da uma chamada de função. 24
3.3 Ilustração da pilha auxiliar de Lua. 25
3.4 Função Lua que retorna um iterador. 26
3.5 Possíveis configurações para instruções portáteis. 27
3.6 Lista de instruções portáteis de Lua. 28
3.7 Acima, uma pequena função em Lua e, abaixo, a saída do programa `luac`. 29

4.1 Implementação da Operação de Soma em LLL. 35
4.2 Modelo de Classes da Representação de Valores em LLL. 36
4.3 Exemplo de uso da API de LLL. 39
4.4 Documentação da interface externa de LLL. 39

5.1 Programas usados nos testes de desempenho. 40
5.2 Número de instruções portáteis em cada programa de teste. 41
5.3 Testes de desempenho medindo o tempo de execução. O tempo de execução é relativo à Lua (Lua = 1). 42
5.4 Tempo de compilação em LLL para cada abordagem. O tempo é relativo à primeira versão (LLL v1 = 1). 43
5.5 Comparação do tempo de execução entre LLL e LuaJit. O tempo de execução é relativo à LuaJIT (LuaJIT = 1). 44
5.6 Comparação do pico de consumo memória entre Lua, LLL e LuaJit. O pico é relativo à LuaJIT (LuaJIT = 1). 45

Que o homem não aprende muito com as lições da história é a mais importante de todas as lições que a história tem para ensinar.

Aldous Huxley

1
Introdução

Linguagens de extensão oferecem mecanismos de alto nível que auxiliam programar de forma mais produtiva. Tipagem dinâmica, coletores de lixo e estruturas de dados nativas permitem escrever código mais expressivo em menos linhas. Dentre essas linguagens, Lua se destaca por ser simples, eficiente e portátil [1]. Seus tipos estão ligados a valores, ela possui coleta de lixo e, ainda, utiliza tabelas como sua única e poderosa estrutura de dados. Além dessas funcionalidades, sua implementação conta com um interpretador, que permite executar o código sem compilá-lo previamente. (Remover a etapa de compilação permite que o processo de desenvolvimento seja consideravelmente acelerado.)

1.1
Definição do Problema

Os mecanismos de alto nível resultam na perda considerável de desempenho durante a execução. Um dos principais problemas é a interpretação do código fonte, que possui um grande custo computacional. Para amenizar tal problema, o interpretador de Lua compila o código fonte em uma representação portátil de instruções, chamada de Lua *bytecode* [2]. Essa compilação é feita durante a execução, de forma eficiente para não impactar no desempenho. Uma vez que as instruções são geradas, elas são executadas pela máquina virtual de Lua (Lua VM).

Ainda que haja grande melhora em relação à interpretação direta do código fonte, muitas otimizações deixam de ser feitas nesta compilação. Realizar essas otimizações consiste num processo demorado e tornar-se-ia o gargalo da execução. Ademais, o uso de uma máquina virtual compromete ainda mais o desempenho em relação ao processador nativo.

Porém, para melhorar o desempenho de Lua é necessário investir em outras formas de executar o código fonte. Uma dessas formas é gerar dinamicamente código de máquina otimizado, utilizando a técnica *just-in-time* (JIT) [3]. Nessa técnica, alguns trechos do programa são compilados durante a execução do interpretador. Heurísticas são aplicadas para definir quais tre-

chos devem ser compilados, já que esse processo é demorado. Essas heurísticas consistem em identificar *hotspots*, trechos de código que são muito executados.

Um dos principais desafios de um JIT é a geração de código de máquina, pois ele é diferente para cada arquitetura. Essa geração, todavia, pode ser feita com o auxílio da cadeia de ferramentas LLVM. A LLVM fica responsável pelas otimizações e pela geração do código de máquina. Assim, é possível implementar um JIT portátil que dá suporte a diversas arquiteturas.

1.2 Soluções Existentes

Terra e Ravi são linguagens baseadas em Lua que utilizam a LLVM para gerar código de máquina [4, 5]. Contudo, elas modificam a especificação original de Lua, inserindo, por exemplo, tipagem estática. Essa abordagem possui desvantagens, pois o desenvolvedor é obrigado a aprender uma nova linguagem. Outro grande problema é a perda da compatibilidade com o interpretador oficial de Lua.

LuaJIT realiza compilação JIT para diversas arquiteturas de processadores e é considerada uma das implementações mais rápidas de linguagens dinâmicas [6]. Porém, tal implementação não utiliza ferramentas intermediárias para a geração de código de máquina. Consequentemente, demasiado esforço é necessário para incluir suporte a novas arquiteturas.

Além do esforço para geração de código de máquina, LuaJIT possui uma máquina virtual própria, com instruções portáteis diferentes das já existentes em Lua. Essa abordagem permite ainda mais ganho de desempenho, mas requer modificações no *parser*, aumentando a complexidade da implementação. Como consequência dessa complexidade, LuaJIT suporta apenas a versão 5.1 de Lua, enquanto a atual é a 5.3.

1.3 Proposta e Objetivos do Projeto

Este projeto tem como objetivo a implementação de um compilador JIT para Lua utilizando a cadeia de ferramentas LLVM. O compilador implementado é uma extensão do interpretador oficial de Lua. Ele provê uma interface para que o usuário final possa escolher quais funções ele desejada compilar durante a execução do programa. Poucas mudanças foram feitas no código original do interpretador e a especificação da linguagem não foi alterada.

Para efetuar a implementação do compilador, primeiro nós realizamos um estudo sobre a cadeia de ferramentas LLVM. Como subtarefa desse estudo, modificamos um compilador já existente para ele que utilizasse a cadeia LLVM.

O compilador modificado foi originalmente implementado durante a disciplina de compiladores (código INF1715, cursada no 7º período).

Após a LLVM, estudamos o funcionamento interno de Lua. A medida que esse estudo foi feito, nós implementamos um protótipo de um compilador de Lua *bytecode* para a representação intermediária de LLVM. Com essa implementação, pudemos confirmar a viabilidade do projeto. Então, com base no protótipo, implementamos a versão final do compilador proposto.

Após essa implementação, realizamos testes de desempenho comparando nosso compilador e o interpretador original Lua. Esses testes avaliaram o tempo de execução, que foi reduzido consideravelmente. Não houve enfoque em reduzir o tempo gasto com as otimizações e a geração de código de máquina. Apresentamos também comparações com LuaJIT, que é considerada o atual estado da arte em termos de desempenho.

O projeto foi desenvolvido nas linguagens C e C++, num sistema Linux, em um computador de arquitetura x86-64. Porém, dada a portabilidade de Lua e de LLVM, ele pode ser facilmente adaptado para outras arquiteturas. Além da LLVM, nenhuma outra biblioteca que não seja padrão da linguagem foi utilizada. A versão do interpretador de Lua modificado foi a 5.3 e a versão utilizada da LLVM foi a 3.6.

Por fim, não alteramos a especificação original de Lua, porém corrotinas não foram implementadas. Por mais que essa seja uma importante funcionalidade de Lua, corrotinas dependem da manipulação da pilha de execução, o que não é provido pela LLVM. Alternativas para tratar corrotinas existem, mas fogem do escopo do projeto e por isso não foram exploradas.

1.4 Estrutura do Relatório

O capítulo 2 introduz ao leitor os principais conceitos da LLVM, sua representação intermediária e sua interface. Será descrito também a principal atividade dos estudos realizados, que consistiu na modificação de um compilador para utilizar a interface C de LLVM.

O capítulo 3 apresenta os principais componentes internos de Lua, em especial as estruturas de dados que representam os valores, as instruções portáteis e a máquina virtual. A medida que foram realizados estudos sobre tais componentes, implementou-se um protótipo do compilador JIT proposto.

O capítulo 4 detalha as principais decisões tomadas durante o projeto e as seções do compilador proposto: a arquitetura, a implementação, a interação com usuário e os testes de qualidade.

O capítulo 5 apresenta os testes de desempenho e os resultados obti-

dos em comparação com Lua VM e LuaJIT. Por fim, o capítulo 6 expõe as considerações finais, principais contribuições e trabalhos futuros.

2
Cadeia de Ferramentas LLVM

LLVM é uma cadeia de ferramentas que, dentre diversas aplicações, permite a geração portátil de código de máquina otimizado. Para isso, a LLVM utiliza uma representação intermediária, chamada de Intermediate Representation (IR). Essa representação é o meio de comunicação entre o *frontend* e o *backend* do compilador [7]. O *frontend* analisa a linguagem fonte e gera a representação intermediária. A LLVM provê ferramentas para otimizar e transformar a IR. Então, o *backend* gera o código de máquina para a arquitetura específica. A figura 2.1 ilustra tais etapas.

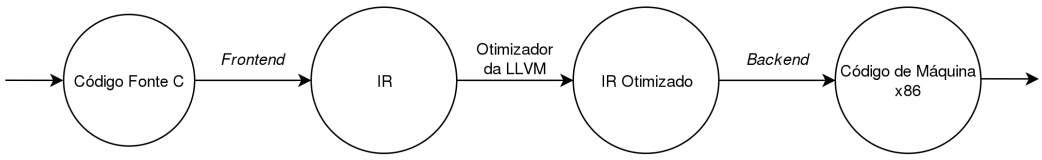

Figura 2.1: Etapas de um compilador de C para x86 que usa LLVM.

Existem *backends* disponíveis e amplamente usados para diversas arquiteturas. Logo, o principal motivo para utilizar a LLVM é a possibilidade de gerar código de máquina de forma portátil. Além dessa vantagem, existe também a etapa de otimização, que é realizada pela LLVM e melhora consideravelmente o desempenho do código gerado.

2.1
Representação Intermediária e Forma SSA

A representação intermediária de LLVM possui três formatos: um textual legível; um interno, manipulado via código; e um binário compactado. Eles são equivalentes entre si, já que a LLVM provê programas e bibliotecas que permitem a conversão de um para outro. Nos artefatos implementados, apenas o formato interno foi utilizado. Contudo, o formato textual foi de extrema importância para o estudo da representação intermediária.

```
1 int foo(int a, int b, int c, int d) {
2     return (a + b) * (c / d);
3 }
```

```
1 define i32 @foo(i32 %a, i32 %b, i32 %c, i32 %d) {
2   %1 = add i32 %b, %a
3   %2 = sdiv i32 %c, %d
4   %3 = mul i32 %2, %1
5   ret i32 %3
6 }
```

Figura 2.2: Acima, uma função simples em C e, abaixo, a função IR gerada pelo compilador Clang.

Além do manual de referência, uma boa forma de estudar a geração de IR é utilizando o compilador Clang [8]. Ao passar os argumentos `-S -emit-llvm`, o compilador emite a representação intermediária com base no código C de entrada. Recomendamos também o uso do argumento `-O2`, para que a representação emitida seja otimizada.

A figura 2.2 apresenta a tradução para IR de uma função em C que retorna o resultado da expressão `(a + b) * (c / d)`. A primeira linha mostra a função em IR, que semanticamente é similar a C. Por outro lado, a sintaxe é bem diferente. Todos os identificadores de variáveis globais começam com o caractere `@` e os de variáveis locais com `%`. As linhas subsequentes contêm as instruções necessárias para computar o valor da expressão. A medida que a expressão é calculada, os valores intermediários são salvos em temporários, enumerados automaticamente pelo Clang. Por fim, há uma instrução que retorna o valor resultante da expressão.

Com base no formato textual, é possível observar que IR possui semelhanças com um conjunto de instruções de três endereços. Outra semelhança está no uso de instruções *jump* para controlar o fluxo de execução do programa. A definição e chamada de funções, por outro lado, é similar à linguagem C, permitindo a abstração dos detalhes de cada arquitetura.

A principal diferença entre IR e um conjunto de instruções está no uso da forma SSA (Single Static Assignment). Essa forma impõe que todas as variáveis temporárias só podem receber uma única atribuição. Consequentemente, a geração automática de IR com base em linguagens procedurais não é trivial [9]. Entretanto, a forma SSA permite que muitas otimizações sejam feitas numa etapa intermediária, sem que haja conhecimento das linguagens fonte e alvo.

Outra característica de IR é a definição obrigatória de tipos em temporários. Essa característica é reforçada ao não permitir conversões implícitas em instruções. A tipagem forte permite que ainda mais otimizações sejam feitas numa etapa intermediária; ela facilita a depuração durante a implementação de um *frontend*, pois muitos erros podem ser encontrados com uma análise estática; e ela auxilia a abstrair definições de arquiteturas específicas.

Para modificar o fluxo de execução de um programa na forma SSA, são necessárias instruções de *jump* que ligam os diferentes blocos básicos. Blocos básicos nada mais são que uma sequência de instruções. Contudo, cada bloco só pode ter um ponto de entrada e um de saída. Logo, não é possível alterar o fluxo de execução após seu início e antes de seu término. Em IR, blocos são identificados por rótulos, que podem ser definidos implicitamente. Se for este o caso, o rótulo é numerado automaticamente, assim como valores temporários.

A figura 2.3 exemplifica o controle de fluxo ao declarar uma função que toma diferentes caminhos com base em seu argumento de entrada. Na primeira linha da representação intermediária, a função `@foo` é declarada de forma análoga a um protótipo em C. Na linha 3, a função `@bar` é definida. Seu bloco de entrada possui o rótulo implícito `%0`, numerado automaticamente. Nesse bloco, a instrução `br` transfere o controle para o bloco `%truepath` caso a condição seja verdadeira. Caso contrário, o bloco `%falsepath` é executado. Além do uso dos blocos, na linha 7 é possível observar que chamadas a funções são feitas de forma similar à linguagem C.

Durante a implementação um *frontend* para IR, representar variáveis mutáveis em SSA é um grande desafio. Cada variável pode ser definida por diversos valores temporários, um para cada atribuição que ela recebe. No entanto, uma instrução especial é necessária para as atribuições que dependem do fluxo de execução. Esse é o caso das que ocorrem dentro de um laço, por exemplo. Essa instrução especial é denominada *phi*. Ela recebe como argumento de entrada um vetor de pares que relacionam valores a blocos básicos. *Phi* retorna o valor associado ao bloco que foi predecessor ao corrente durante a execução.

A figura 2.4 apresenta o uso da instrução *phi* na representação da variável de controle de um laço. Na linha 8, dois pares são passados como argumento de entrada para a instrução *phi*. O primeiro associa o valor 0 ao bloco de entrada, e o segundo associa o temporário `%increment` ao bloco `%loopbody`. Durante a execução, o temporário `%i` recebe o valor associado ao último bloco executado. Logo, caso bloco de entrada tenha sido o predecessor, `%i` recebe o valor 0. Caso contrário, `%i` é incrementado.

Finalmente, o último aspecto de IR importante para este trabalho é a

```
int foo();

int bar(bool condition) {
  if (condition) {
    return foo();
  } else {
    return -1;
  }
}
```

```
declare i32 @foo()

define i32 @bar(i1 %condition) {
  br i1 %condition, label %truepath, label %falsepath

truepath:
  %x = call i32 @foo()
  ret i32 %x

falsepath:
  ret i32 -1
}
```

Figura 2.3: Acima, uma função em C com uma condicional. Abaixo, uma função IR equivalante que exemplifica o uso de blocos básicos. Escrevemos essa função em IR manualmente ao invés de utilizar o compilador Clang.

```
void bar(int n) {
  for (int i = 0; i < n; i++) {
    foo();
  }
}
```

```
define void @bar(i32 %n) {
entry:
  br label %loopcheck

loopcheck:
  %i = phi i32 [ 0, %entry ], [ %increment, %loopbody ]
  %i_lt_n = icmp slt i32 %i, %n
  br i1 %i_lt_n, label %loopbody, label %exit

loopbody:
  call void @foo()
  %increment = add i32 %i, 1
  br label %loopcheck

exit:
  ret void
}
```

Figura 2.4: Acima, uma função em C com um laço *for*. Abaixo, a função em IR equivalente.

definição de estruturas. Assim como funções, tal definição é semelhante à linguagem C. Contudo, o acesso aos seus campos é feito pela instrução GEP (Get Element Pointer). Essa instrução recebe dois índices como argumentos de entrada. O primeiro índice indica a posição do elemento no vetor, e o segundo especifica qual campo do elemento será acessado. Em C, `&v[i].field` seria a expressão equivalente a essa instrução, no qual `v` é o vetor de estruturas, `i` o índice do elemento e `field` o nome do campo. Para acessar um campo de uma estrutura que não está em um vetor, basta obter o endereço da estrutura e passar 0 como o primeiro índice da instrução GEP.

2.2 Interface com C e C++

As diversas bibliotecas que compõem a cadeia LLVM são escritas na linguagem C++. A principal interface de LLVM é nessa mesma linguagem, mas existe também uma interface na linguagem C. A interface em C é uma adaptação da original em C++, e ela possui apenas um subconjunto de todas as funcionalidades de LLVM. Contudo, esse subconjunto abrange as ferramentas necessárias para a criação de um compilador estático. Além disso, a interface em C é considerada mais estável, pois não sofre tantas mudanças quanto a em C++ [10].

Uma característica marcante da interface em C++ é a necessidade de manipular diversos objetos. Na interface em C, as classes que definem tais objetos são substituídas por tipos abstratos de dados. Internamente, as funções da interface em C fazem chamadas aos métodos das classes em C++. Dessa forma, o uso de ambas as interfaces é bem similar.

Durante este projeto, a falta de documentação foi o maior problema encontrado ao utilizar a interface de LLVM [11]. Recorremos a tutoriais disponibilizados por fontes não oficiais para o seu entendimento inicial [12, 13]. Com base nesses tutoriais, a implementação de um compilador estático demonstrou-se relativamente simples. O compilador estático que implementamos será descrito na seção 2.3.

Contudo, ainda que tais tutorias tenham sido úteis, eles não continham todo o conjunto de funcionalidades necessárias para a implementação de um compilador JIT. Em especial, na versão 3.6 de LLVM, há um *bug* que impossibilita a chamada de funções externas dentro do código gerado dinamicamente [14]. (Uma função é denominada externa quando ela é definida no programa hospedeiro, e não em IR.)

A interface em C++ provê uma forma de contornar tal *bug*. Contudo, pela falta de documentação, a implementação de chamadas externas foi um

```
1 int fatorial(int n) {
2     int r;
3     r = 1;
4     while (n > 1) {
5         r = n * r;
6         n = n - 1;
7     }
8     return r;
9 }
```

Figura 2.5: Função fatorial em Monga.

grande desafio. Dessa forma, apresentamos no apêndice A um pequeno programa em C++ que contorna esse *bug*. Entretanto, a interface em C não provê a funcionalidade necessária para realizar tal implementação. Como essa é uma funcionalidade essencial para o compilador JIT de Lua, nós realizamos uma transição para a interface C++.

2.3 Modificação do Compilador de Monga

Durante o estudo sobre LLVM e IR, uma importante tarefa foi a modificação de um compilador da linguagem Monga para que ele utilizasse a interface C de LLVM. A linguagem Monga consiste num subconjunto reduzido de C. Ela foi a linguagem utilizada durante a disciplina de compiladores, e seu compilador foi implementado durante essa disciplina. O compilador original de Monga gera código *assembly* para a arquitetura IA-32.

2.3.1 Compilador Original de Monga

Monga possui expressões aritméticas, estruturas de controle, definição de funções e vetores alocados dinamicamente. Outros recursos, como ponteiros e outras estruturas de dados, foram propositalmente removidos da linguagem para redução do escopo. Por mais que a linguagem seja limitada, ela possui os elementos necessários para o aprendizado do funcionamento de um compilador. A figura 2.5 apresenta como exemplo a função fatorial em Monga.

O compilador original de Monga possui quatro etapas: as análises léxica, sintática e semântica e a geração do código alvo. As análises léxica e sintática são feitas com as ferramentas Lex e Yacc, respectivamente [15, 16]. A medida que tais análises são feitas, a árvore sintática é gerada de acordo com o código fonte. Após a geração de toda à árvore, a análise semântica faz a amarração

das variáveis e verifica a coerência dos tipos. Por fim, a árvore é percorrida e a geração do código alvo é feita.

O módulo de geração do código alvo declara externamente apenas uma função. Essa função recebe a árvore sintática como argumento de entrada e imprime o código alvo na saída padrão. Então, a saída padrão é redirecionada para um arquivo e um assembler era utilizado para gerar o executável final.

2.3.2 Modificações Feitas

Como o compilador tinha sido dividido corretamente em módulos, bastou substituir a geração de *assembly* IA-32 pela geração da representação intermediária de LLVM. Na mudança que fizemos, ao invés de imprimir diretamente a forma textual de IR, utilizamos o formato interno. Após a geração de toda a representação, o formato interno é convertido para o formato textual pela própria LLVM. Por fim, a saída é redirecionada para um arquivo, o programa em IR é otimizado pela ferramenta `opt` e um executável é gerado com a `llc`. (As ferramentas `opt` e `llc` são parte da distribuição de LLVM.)

A geração de IR é similar a de *assembly*, e por isso a maior parte da implementação foi trivial. Contudo, a geração de código na forma SSA foi um grande desafio. Para representar as variáveis de Monga, foi criada uma tabela que as relaciona à valores temporários de IR. Durante a compilação, a medida que atribuições são feitas, essa tabela é atualizada com o último valor atribuído.

A implementação correta das instruções *phi* foi feita usando o algoritmo proposto por Brandis e Mössenböck [17]. Como Monga é uma linguagem estruturada e não possui declarações *goto*, foi possível gerar código na forma SSA sem necessidade de algoritmos complexos e diversas estruturas de dados auxiliares. A figura 2.6 expõe a saída do compilador modificado para a função fatorial da figura 2.5.

2.3.3 Resultados Obtidos

Após a implementação, fizemos testes de desempenho com o intuito de avaliar o tempo de execução dos executáveis gerados. Como base para as comparações, criamos programas equivalentes em C, e os compilamos com GCC e Clang. Com o intuito de otimizar a saída do compilador de Monga, utilizamos a ferramenta `opt` com o argumento -O2. Este mesmo argumento foi passado para os outros compiladores.

Os programas medidos foram o cálculo recursivo dos 40 primeiros nú-

```
define i32 @fatorial(i32 %n) {
entry:
    %0 = icmp sgt i32 %n, 1
    br i1 %0, label %loop_in, label %loop_end

loop_in:
    %1 = phi i32 [ %n, %entry ], [ %4, %loop_in ]
    %2 = phi i32 [ 1, %entry ], [ %3, %loop_in ]
    %3 = mul i32 %1, %2
    %4 = sub i32 %1, 1
    %5 = icmp sgt i32 %4, 1
    br i1 %5, label %loop_in, label %loop_end

loop_end:
    %6 = phi i32 [ %n, %entry ], [ %4, %loop_in ]
    %7 = phi i32 [ 1, %entry ], [ %3, %loop_in ]
    ret i32 %7
}
```

Figura 2.6: IR gerado pelo compilador de Monga para a função fatorial da figura 2.5.

meros da sequência de Fibonacci, a multiplicação de duas matrizes 750×750 e a ordenação de um vetor de 25 mil elementos. Fizemos a média do tempo de 10 execuções para cada programa. Cada execução durou aproximadamente 1 segundo.

A figura 2.7 apresenta os resultados obtidos para cada compilador e programa. O eixo vertical do gráfico indica a razão do tempo de execução do executável em relação ao equivalente gerado por Monga. No programa que calcula a sequencia de Fibonacci, observamos que apenas o GCC gerou um executável mais rápido. Na multiplicação de matrizes, todos os compiladores geraram executáveis equiparáveis. Na ordenação, apenas Clang se sobressaiu.

Com base nos resultados obtidos, notamos que o compilador de Monga gera executáveis equiparáveis aos de GCC e Clang. Isso é possível por conta da representação intermediária e todo o conjunto de ferramentas disponibilizado pela LLVM. Vale ressaltar que o módulo de geração de IR possui apenas 1408 linhas de código e foi desenvolvido em um mês. Concluímos então que o uso da LLVM permite a implementação de compiladores eficientes sem a grande complexidade presente na abordagem tradicional.

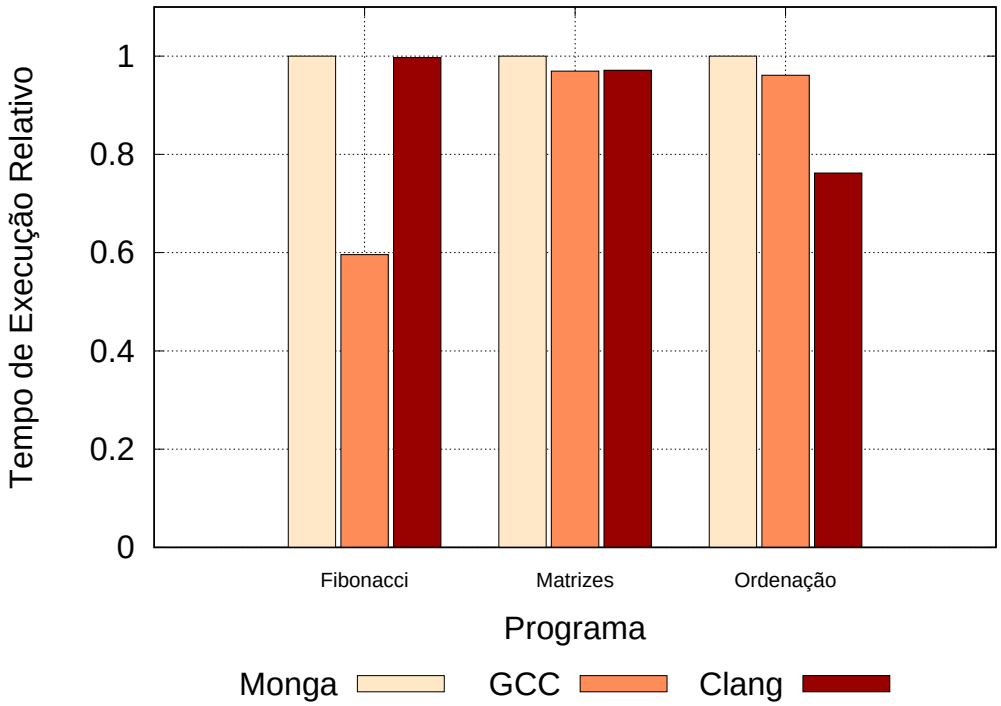

Figura 2.7: Comparações dos tempos de execução em relação aos executáveis gerados por Monga.

3
Componentes Internos do Interpretador de Lua

Lua é uma linguagem de extensão que se destaca por ser rápida e portátil. O bom desempenho é uma das consequências da compilação do código fonte para uma representação portátil. A portabilidade vem da implementação do interpretador na linguagem C, repeitando o padrão ANSI.

Internamente, o interpretador de Lua é subdividido em diversos módulos. A implementação da máquina virtual, por exemplo, é exclusiva ao módulo `lvm`. Consequentemente, todo o processamento das instruções portáteis é independente da análise do código fonte. Dessa forma, foi possível estudar apenas um subconjunto dos módulos existentes ao efetuar este projeto.

O estado Lua é o ponto inicial para compreender o funcionamento interno. A estrutura que o representa é declarada como `lua_State`, no módulo `lstate`. O objetivo do estado é armazenar todas as outras estruturas necessárias para a execução do interpretador. A execução depende apenas de informações provenientes de um único estado, e diferentes estados não podem se comunicar diretamente. Não são utilizadas variáveis globais na implementação de Lua.

As seções seguintes descreverão os componentes internos de Lua mais relevantes para este trabalho. Em especial, será detalhada a representação dos valores, as instruções portáteis e a máquina virtual. Serão apresentados apenas os aspectos internos e assume-se que o leitor possui conhecimento prévio da linguagem.

3.1
Representação de Valores

Em Lua, tipos são ligados a valores e não a variáveis. Eles podem ser nulos, números, booleanos, *strings*, funções, tabelas, dados do usuário ou corrotinas. A estrutura que os representa é denominada `TValue`, declarada no módulo `lobject`. Essa estrutura possui dois campos: uma enumeração para o tipo, também chamado de rótulo; e uma *union* para o valor em si.

Os valores de tipos básicos são armazenados na própria *union*, pois ocupam pouco espaço em memória. Já os que precisam de estruturas de dados

auxiliares são representados por ponteiros. Nesse caso, o valor é apenas uma referência para o objeto alocado em outro lugar. Esses objetos são coletados automaticamente quando não há mais referências para eles.

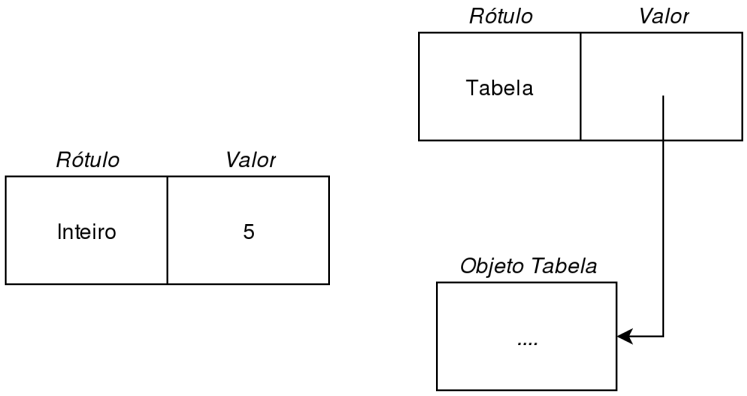

Figura 3.1: Exemplos de valores em Lua.

A figura 3.1 apresenta dois valores de Lua como exemplo. À esquerda, há um valor do tipo inteiro, logo seu conteúdo é armazenado internamente. À direita, há um valor cujo conteúdo é apenas uma referência para o objeto que representa a tabela em si.

3.2 Pilha de Registradores

A máquina virtual de Lua opera sobre os registradores que representam as variáveis locais e temporários de uma função. Esses registradores são armazenados numa pilha, implementada como um vetor de valores. A utilização de um vetor é essencial, pois as instruções portáteis podem alterar registradores em qualquer posição da pilha, não apenas no topo. As instruções portáteis serão detalhadas na seção 3.4.

De forma similar à pilha utilizada na implementação de C, a pilha de Lua cresce a medida que funções são chamadas. Inicialmente, a função a ser chamada e seus argumentos de entrada são empilhados. Então, a função é chamada e a máquina virtual executa as instruções portáteis, modificando os registradores da pilha. Por fim, os valores retornados são movidos para as posições em que estavam a função chamada e os argumentos. O módulo `ldo` é o responsável por alterar a pilha e o estado Lua quando uma chamada é feita.

A figura 3.2 ilustra o processo de chamada e retorno de uma função. No estado inicial *a*, o topo da pilha é um valor qualquer. Em *b*, a função denominada `soma` e os argumentos 3 e 5 são empilhados. Em *c*, a função é

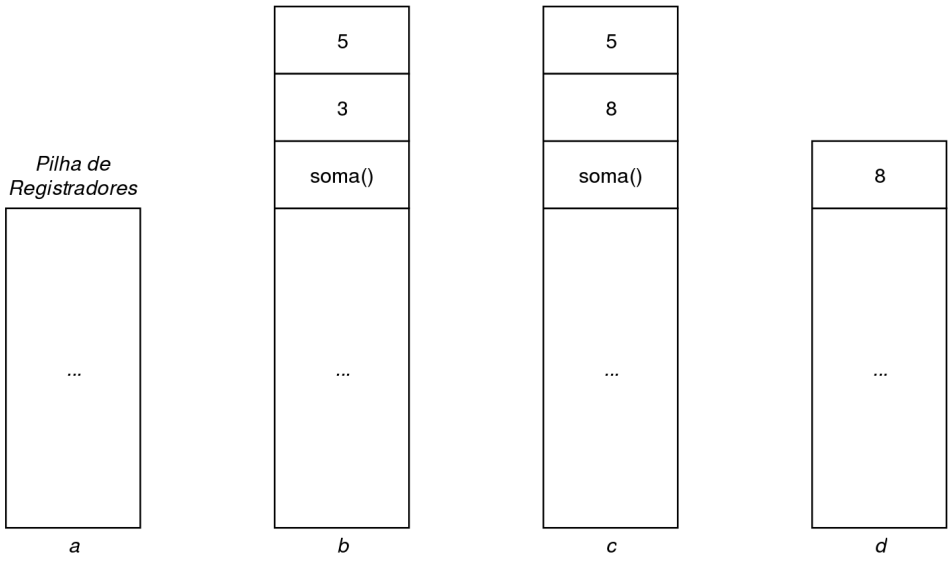

Figura 3.2: Exemplo das etapas da uma chamada de função.

executada e o resultado da adição é atribuído ao registrador que possuía a valor 3. Finalmente, no estado *d*, o valor de retorno é movido para o registrador que armazenava a função `soma`.

Nesta figura, o valor da função executada foi representado por um nome seguido de um par de parênteses. Contudo, em Lua, todas as funções são objetos anônimos e não possuem um nome associado. Essa representação foi escolhida apenas para facilitar o entendimento da figura. A implementação das funções será apresentada na seção 3.3.

Além da pilha de registradores, outra pilha é necessária para armazenar a posição de cada função chamada. Sem essa pilha auxiliar, não seria possível saber para onde os valores retornados devem ser movidos. A segunda pilha é implementada como uma lista encadeada, que possui instâncias da estrutura `CallInfo`. Esta estrutura é declarada no módulo `lstate`.

A figura 3.3 exemplifica a interação entre essas duas pilhas. Nessa figura, estão presentes duas chamadas, uma à função `bar` e outra à `foo`. Para cada chamada, existe uma instância de `CallInfo` na pilha auxiliar. E cada instância possui um campo que referencia os registradores que armazenam estas duas funções.

Para cada instância de `CallInfo`, além da referência à função, é necessária uma outra referência para o registrador base de cada chamada. Esse registrador indica a posição da primeira variável local ou temporário utilizado pela função. Na figura 3.2, por exemplo, o registrador base é o sucessor do

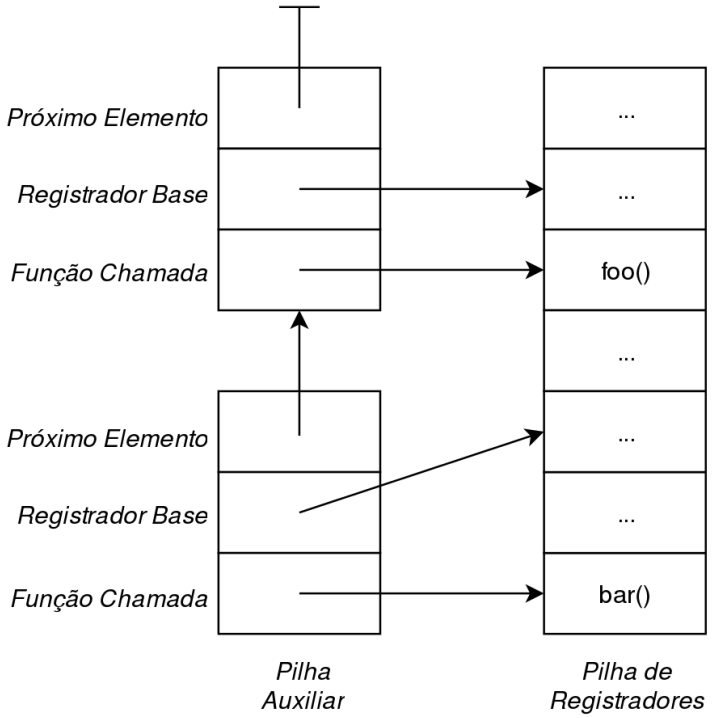

Figura 3.3: Ilustração da pilha auxiliar de Lua.

que armazena a função chamada. Isso ocorre apenas quando a função possui um número fixo de argumentos de entrada. Nesse caso, os registradores que armazenam os argumentos são utilizados também como temporários.

Em funções variádicas, os argumentos de entrada são armazenados após a função e antes do registrador base. Dessa forma, a referência para base é definida apenas no momento da execução. Na figura 3.3, a função `bar` é variádica e nesta chamada recebe um único argumento.

3.3 Funções

Em Lua, todas as funções são anônimas, ou seja, não estão associadas a nomes ou variáveis. Diferentemente de linguagens estáticas como C, funções são valores de primeira classes. Logo, funções podem ser atribuídas a variáveis, passadas como parâmetros e retornadas por outras funções.

Duas estruturas são necessárias para a implementação de funções. A primeira é o protótipo, no qual as instruções portáteis e as constantes são armazenadas. A segunda é denominada *closure*, que referencia o protótipo e armazena as variáveis do ambiente externo usadas pela função (em Lua, essas variáveis são conhecidas como *upvalues*). Essas estruturas são definidas no

```
1 function create_iterator()
2   local i = 0
3   return function()
4     i = i + i
5     return i
6   end
7 end
```

Figura 3.4: Função Lua que retorna um iterador.

módulo `lobject` como `Proto` e `LClosure` respectivamente.

Durante a interpretação do código fonte de Lua, as funções são compiladas em instruções portáteis. Essa compilação é realizada uma única vez e seu resultado é armazenado no protótipo. Além das instruções portáteis, é gerada também uma tabela de constantes para cada função. Ao invés de armazenar as constantes dentro de instruções, elas ficam numa tabela. Durante a execução, a máquina virtual acessa as constantes na tabela com base em seus índices.

Um novo *closure* é criado cada vez que a função é instanciada durante a execução. *Closures* são necessários pois os ambientes não são compartilhados pelas funções de mesmo protótipo. A figura 3.4 ilustra esse comportamento. Um novo *closure* é criado cada vez que a função `create_iterator` é chamada. A função anônima criada na linha 3 possui um *upvalue*, que é representado pela variável local `i`.

3.4 Instruções Portáteis

As instruções portáteis de Lua são representadas por inteiros sem sinal de 32 bits. Cada inteiro é subdividido em alguns campos, que armazenam as informações necessárias para execução da instrução. Para ler e alterar esses campos, existem funções e macros que encapsulam as operações de manipulação de bits. Essas funções são definidas no módulo `lopcodes`.

O primeiro campo é denominado `OpCode`, pois armazena o código da operação. Esse código indica qual operação deve ser realizada pela máquina virtual. Carregar uma constante ou somar dois números são exemplos de operações. Exitem 47 possíveis operações, declaradas numa enumeração no módulo `lopcodes`. A figura 3.6 apresenta a documentação dessas 47 operações.

A configuração dos outros campos, também chamados de argumentos, depende da operação realizada. A figura 3.5 apresenta as possíveis configurações para uma instrução portátil. Os campos `A`, `B`, `C`, `Ax` e `Bx` são inteiros sem sinal e o campo `sBx` é um inteiro com sinal.

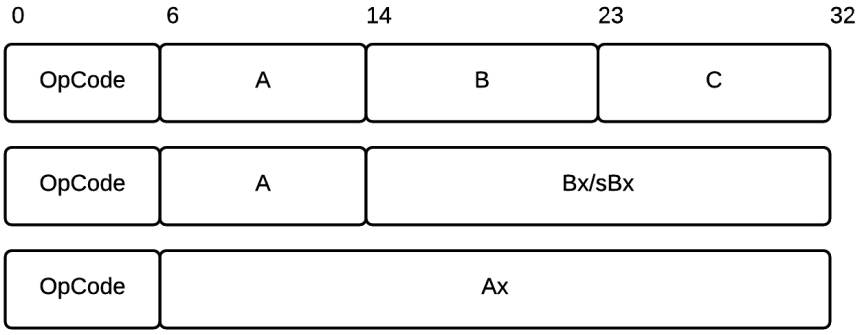

Figura 3.5: Possíveis configurações para instruções portáteis.

Assim como a configuração dos campos, a semântica de cada campo depende da operação. Na operação de adição, por exemplo, são utilizados os campos `A`, `B` e `C`. O campo `A` denota em qual registrador será armazenado o resultado. Os campos `B` e `C` são referências para constantes ou registradores utilizados como operandos da adição.

Com intuito de indicar quais registradores são manipulados, a documentação no código do interpretador de Lua utiliza a notação `R(x)`. Essa notação evidencia a macro utilizada para acessar o registrador de índice `x`. A indexação utilizada na macro é iniciada no registrador base de cada função. Logo, `R(0)` representa o registrador base, `R(2)` o segundo registrador após o base e `R(A)` o registrador de índice `A`.

De forma análoga a registradores, existe a macro `K(x)` utilizada para obter constantes. Como explicado na seção 3.3, cada função Lua possui sua própria tabela de constantes. Logo, o parâmetro `x` da macro é o índice da constante na tabela. Essa abordagem permite as constantes sejam armazenadas fora das instruções, possibilitando a economia de memória. Diversas constantes são do tipo *double* ou *string*, que ocupam por si só mais que o dobro do espaço de uma única instrução.

Além das macros `R(x)` e `K(x)`, existe a `RK(x)`. Essa macro permite que registradores e constantes sejam utilizadas de forma intercambiável. Se o primeiro bit do argumento `x` estiver ligado, esse bit é eliminado e a macro `K(x)` é chamada. Caso contrário, o registrador correspondente é obtido pela macro `R(x)`. Essa abordagem permite a utilização direta de constantes por algumas instruções, sem a necessidade de primeiro carregar a constante em um registrador. (Essa característica é conhecida como operando imediato em assembly).

Instrução	Descrição
MOVE	R(A) := R(B)
LOADK	R(A) := K(Bx)
LOADKX	R(A) := K(extra arg)
LOADBOOL	R(A) := (Bool)B; if (C) pc++
LOADNIL	R(A), R(A+1), ..., R(A+B) := nil
GETUPVAL	R(A) := UpValue[B]
GETTABUP	R(A) := UpValue[B][RK(C)]
GETTABLE	R(A) := R(B)[RK(C)]
SETTABUP	UpValue[A][RK(B)] := RK(C)
SETUPVAL	UpValue[B] := R(A)
SETTABLE	R(A)[RK(B)] := RK(C)
NEWTABLE	R(A) := { (size = B,C) }
SELF	R(A+1) := R(B); R(A) := R(B)[RK(C)]
ADD	R(A) := RK(B) + RK(C)
SUB	R(A) := RK(B) - RK(C)
MUL	R(A) := RK(B) * RK(C)
MOD	R(A) := RK(B) % RK(C)
POW	R(A) := RK(B) ^ RK(C)
DIV	R(A) := RK(B) / RK(C)
IDIV	R(A) := RK(B) // RK(C)
BAND	R(A) := RK(B) & RK(C)
BOR	R(A) := RK(B) | RK(C)
BXOR	R(A) := RK(B) ~ RK(C)
SHL	R(A) := RK(B) << RK(C)
SHR	R(A) := RK(B) >> RK(C)
UNM	R(A) := -R(B)
BNOT	R(A) := ~R(B)
NOT	R(A) := not R(B)
LEN	R(A) := length of R(B)
CONCAT	R(A) := R(B).. ... ..R(C)
JMP	pc+=sBx; if (A) close all upvalues >= R(A - 1)
EQ	if ((RK(B) == RK(C)) ~= A) then pc++
LT	if ((RK(B) < RK(C)) ~= A) then pc++
LE	if ((RK(B) <= RK(C)) ~= A) then pc++
TEST	if not (R(A) <=> C) then pc++
TESTSET	if (R(B) <=> C) then R(A) := R(B) else pc++
CALL	R(A), ... ,R(A+C-2) := R(A)(R(A+1), ... ,R(A+B-1)
TAILCALL	return R(A)(R(A+1), ... ,R(A+B-1)
RETURN	return R(A), ... ,R(A+B-2)
FORLOOP	R(A)+=R(A+2); if R(A) <= R(A+1) then { pc+=sBx; R(A+3)=R(A) }
FORPREP	R(A)-=R(A+2); pc+=sBx
TFORCALL	R(A+3), ... ,R(A+2+C) := R(A)(R(A+1), R(A+2))
TFORLOOP	if R(A+1) ~= nil then { R(A)=R(A+1); pc += sBx }
SETLIST	R(A)[(C-1)*FPF+i] := R(A+i), 1 <= i <= B
CLOSURE	R(A) := closure(KPROTO[Bx])
VARARG	R(A), R(A+1), ..., R(A+B-2) = vararg
EXTRAARG	extra (larger) argument for previous opcode

Figura 3.6: Lista de instruções portáteis de Lua.

Além da documentação do código fonte, uma importante referência para o estudo das instruções portáteis é o artigo escrito por Man [18]. Esse artigo aborda de forma detalhada as instruções da versão 5.1 de Lua. Apesar da versão apresentada não ser a mais atual, o artigo ainda possui grande relevância, pois poucas alterações foram feitas na definição das instruções.

Para compreender melhor a geração das instruções portáteis, é possível utilizar a ferramenta `luac`. Essa ferramenta, distribuída junto ao interpretador de Lua, gera um arquivo binário com as instruções portáteis de um programa Lua. Contudo, ao passar os argumentos `-l -l`, as instruções geradas são impressas na tela de forma legível.

A figura 3.7 exibe a saída de `luac` para uma pequena função em Lua. A primeira linha da saída apresenta o nome do arquivo de entrada, as linhas em que a função está declarada e o número de instruções. A segunda linha traz o número de parâmetros, registradores, *upvalues*, variáveis locais, constantes e funções aninhadas.

Em seguida, são descritas as quatro instruções que compõem a função. Para cada instrução, é impresso o índice da instrução, a linha em que ela apa-

```
1 function f(array, index)
2   return return array[index + 2]
3 end
```

```
1  function <input.lua:1,3> (4 instructions at 0x2517780)
2  2 params, 3 slots, 0 upvalues, 2 locals, 1 constant, 0
       functions
3          1        [2]     ADD             2 1 -1  ; - 2
4          2        [2]     GETTABLE        2 0 2
5          3        [2]     RETURN          2 2
6          4        [3]     RETURN          0 1
7  constants (1) for 0x2517780:
8          1        2
9  locals (2) for 0x2517780:
10         0        array   1       5
11         1        index   1       5
12 upvalues (0) for 0x2517780:
```

Figura 3.7: Acima, uma pequena função em Lua e, abaixo, a saída do programa `luac`.

rece, o código da operação e seus argumentos. A instrução ADD realiza a adição entre o registrador 1 e a constante 1, atribuindo o resultado ao registrador 2. A instrução GETTABLE acessa a tabela do registrador 0 com o índice armazenado no registrador 2 e salva o resultado neste próprio.

Por fim, é feito o retorno do valor armazenado no segundo registrador. Com intuito de simplificar a compilação, uma instrução que retorna nulo é sempre inserida no final de todas as funções, mesmo que não ela não seja executada. Após a definição das instruções, é impressa também a tabelas de constantes. Por fim, são impressas duas tabelas, uma com os nomes das variáveis locais e outra com os nomes dos *upvalues*. Essas últimas duas tabelas são utilizadas apenas durante a depuração.

3.5 Máquina Virtual

A máquina virtual de Lua é defina no módulo `lvm`. Seu funcionamento consiste num laço que percorre e processa as instruções portáteis da função corrente. Dentro desse laço, há uma declaração *switch* que executa a instrução corrente de acordo com seu código. A implementação das instruções na máquina virtual segue a descrição da figura 3.6. As instruções mais simples, como as de código MOVE, são implementadas dentro do laço principal. Outras mais complexas requerem diversas funções auxiliares e acessam outros módulos de

Lua.

Quando uma chamada de função é feita, a máquina virtual chama o módulo `ldo` para modificar as pilhas. Após as modificações, o controle da execução volta para o laço da máquina virtual. Dessa forma, a pilha de Lua pode crescer sem que a pilha de C também cresça. Logo, há controle sobre o tamanho da pilha de Lua, evitando falhas de estouro, por exemplo.

Outro importante motivo para o desacoplamento das pilhas é a implementação de corrotinas. Cada corrotina possui uma linha de execução própria, consequentemente, precisa de uma pilha própria. Para manter a implementação de Lua portátil, é necessário criar várias pilhas de Lua sem precisar de *threads* do sistema. Como as corrotinas não executam paralelamente, apenas uma *thread* do sistema é necessária para executar a máquina virtual de Lua.

Os módulos que contemplam a implementação específica de outras estruturas não precisaram ser estudados. A manipulação de tabelas, por exemplo, é feita internamente pelo módulo `ltable`. Todas as operações são devidamente encapsuladas, por conseguinte, não é necessário saber detalhes de sua implementação. *Strings* e corrotinas também são encapsulados por seus módulos específicos.

4
Compilador de Lua bytecode para LLVM IR

Este projeto consistiu na geração de uma nova distribuição de Lua, chamada de Lua Low Level (LLL). Esse nome faz alusão à Lua e a LLVM, pois esta já foi a abreviatura de *Low Level Virtual Machine*. LLL é um JIT que compila funções Lua em código de máquina durante a execução do interpretador. Para realizar tal compilação, LLL gera uma função na representação intermediária de LLVM equivalente à função em Lua. Essa função em IR é criada com base nas instruções portáteis de Lua. Após a geração da função em IR, LLL utiliza LLVM para gerar o código de máquina dinamicamente.

Um dos requisitos deste projeto foi fazer o menor número de modificações no código original do interpretador de Lua. Inclusive, nós consideramos a possibilidade de criar uma biblioteca separada. Contudo, dada a necessidade de alterar alguns pontos do interpretador original, criamos um novo executável.

A decisão de quais funções são compiladas é feita pelo usuário final. Esse usuário utiliza uma biblioteca que adicionamos em Lua para compilar uma função. Nós implementamos também uma forma automática de identificar *hotspots* e compilar as funções automaticamente. Funções são consideradas *hotspots* quando são chamadas um determinado número de vezes. Assim que esse número é alcançado, a função é compilada. Entretanto, a identificações de *hotspots* não foi o foco do projeto e por isso não investimos mais esforço nesse tema.

A implementação de LLL teve início em setembro de 2015 e teve duração de 7 meses. No seu decorrer, muitas alterações foram feitas na arquitetura, mas a proposta de modificar pouco Lua se manteve. A versão final de LLL possui 3519 linhas de código, e foram inseridas apenas 28 linhas no interpretador oficial. Este capítulo tem como foco a última versão, contudo ele apresenta também as decisões tomadas durante o desenvolvimento das anteriores.

Dividimos o projeto internamente em alguns módulos. O principal deles interage com o interpretador de Lua e coordena a compilação das funções. Descrevemos essa interação na seção 4.1. Na seção 4.2, descrevemos a compilação das instruções portáteis de Lua. Na seção 4.3, descrevemos a representação de valores de Lua em tempo de compilação. Na seção 4.4, descrevemos os

testes automáticos que avaliam a corretude de LLL. Finalmente, na seção 4.5, descrevemos a interação com o usuário final.

4.1 Interação com Interpretador de Lua

As primeiras versões de LLL foram feitas utilizando a linguagem C. Contudo, durante a implementação, sentimos necessidade de utilizar uma linguagem de mais alto nível. Escolhemos a linguagem C++ porque ela permite a fácil integração com C e possui características que facilitam o desenvolvimento. Em especial, utilizamos estruturas de dados disponibilizadas pelas bibliotecas padrões e o paradigma de orientação a objeto. Além dessas vantagens, a mudança para C++ foi essencial dado o *bug* da interface C de LLVM, descrito no capítulo 2.

Implementamos uma única forma de compilar uma função Lua. Essa forma consiste em percorrer as instruções portáteis da função, gerando a representação intermediária de LLVM. Após a compilação para IR, a LLVM gera o código de máquina para a arquitetura específica. O código gerado é armazenado no protótipo da função, em um novo campo inserido na estrutura `Proto`. Dessa forma, diferentes *closures* que utilizam o mesmo protótipo compartilham o código gerado. Para realizar a chamada do código gerado, modificamos a função `precall` do módulo `ldo`. Adicionamos uma verificação que chama o código gerado caso ele exista.

Para identificar *hotspots*, adicionamos outro campo no protótipo das funções. Esse campo é apenas um contador que é incrementado cada vez que a função é chamada. A compilação automática é realizada assim que esse contador ultrapassa um limiar. Assim como a chamada do código gerado, o incremento do contador e a compilação automática são feitos dentro da função `precall`.

Como visto no capítulo 3, Lua possui diversas estruturas de dados. Essas estruturas devem ser redeclaradas na representação intermediária de LLVM para serem utilizadas na função em IR. Porém, redefinir manualmente todos os campos das estruturas é trabalhoso e propenso a erros. Logo, foi adotada a estratégia de declarar as estruturas em IR com um único campo. Esse campo é um inteiro que possui o tamanho da estrutura, dado pela macro `sizeof`.

Para acessar os campos em IR, primeiro convertemos o ponteiro para a estrutura em um vetor de bytes. Então, utilizamos a macro `offsetof` para saber a posição do vetor que armazena o campo. Por fim, convertemos o ponteiro para essa posição em um ponteiro com o tipo correspondente do campo. Essas conversões são necessárias, pois a representação intermediária

de LLVM é fortemente tipada. Contudo, ao inspecionar o código assembly gerado, observamos que não há penalidade ao utilizar essa estratégia.

Durante a execução do código gerado, diversas estruturas de Lua deverão ser acessadas. Para realizar esse acesso, passamos o estado Lua como o único argumento de entrada para função em IR. Com base no estado Lua, a função em IR poderá obter todas as informações que precisa durante a chamada.

4.2 Compilação das Instruções Portáteis de Lua

Funções Lua são representadas por uma sequência de instruções portáteis. LLL transforma cada uma dessas instruções em pelo menos um bloco básico de LLVM. Essa abordagem permite que cada instrução seja compilada de forma independente. Outra vantagem está na implementação direta da instrução JUMP, pois já se sabe de antemão para qual bloco se deve pular. Vale ressaltar que a etapa de otimização da LLVM remove penalidades de eficiência causadas por essa abordagem.

Mover o valor de um registrador para outro ou carregar uma constante são consideradas instruções simples. Logo, a compilação dessas instruções é direta, bastando algumas linhas de código. Outras instruções são consideradas complexas, pois necessitam diversas condicionais e, consequentemente, de muitos blocos básicos.

Ao compilar as instruções aritméticas, por exemplo, é necessário gerar código que verificará o tipo de cada operando durante a execução. Dependendo dos tipos, diferentes operações poderão ser realizadas. Assim, na versão inicial de LLL, implementamos instruções complexas com chamadas a funções externas.

Funções externas são aquelas chamadas pelo código gerado e compiladas junto à aplicação. Para cada instrução aritmética, implementamos uma função externa em C++. Durante a execução, o código em IR chama a função externa passando os campos da instrução como argumento de entrada. Então, a função externa avalia o resultado e o armazena na pilha de Lua.

Assim como estruturas, as funções externas devem ser redeclaradas utilizando a interface de LLVM. Deve-se definir o protótipo da função em IR e prover um ponteiro para a implementação da função. É importante ter cuidado ao redefinir tais funções, pois a consistência dos tipos fica a cargo do programador. Outra desvantagem é que a expansão *inline* não pode ser feita, já que a função não está em IR.

Utilizar funções externas para compilar instruções complexas propiciou uma redução no tempo de execução em comparação ao interpretador de Lua.

Contudo, os resultados foram abaixo do esperado, e por isso adotamos outra estratégia. As funções externas foram substituídas por trechos de código em IR, removendo assim o gargalo gerado pelas chamadas. Em especial, aplicamos essa estratégia para as instruções aritméticas e as de acesso a tabelas.

Essa tarefa foi trabalhosa, pois IR é uma linguagem de baixo nível. Como exemplo, a figura 4.1 apresenta o fluxo da instrução de adição. Losangos representam pontos de decisão e retângulos as operações. Como não se pode alterar o fluxo durante um bloco básico, são necessários dez blocos para implementar essa instrução. Os tipos dos dois argumentos serão verificados antes de realizar a operação. Além dos tipos, há a coerção de cadeias de caracteres para números, realizada pela função `tonumber`. (Essa função também converte de inteiro para ponto flutuante).

Após a troca de abordagem para a compilação das instruções complexas, adicionamos uma etapa extra de otimização. Em LLVM, há diversas passagens de otimização disponíveis que transformam a representação intermediária gerada. Incluímos passagens para propagar constantes e eliminar código morto. A etapa extra de otimização é necessária porque as etapas já existentes em LLVM são focadas em linguagens estáticas, como C e C++ [19]. Em especial, observamos que a passagem de otimização `mem2reg` não é realizada por padrão. Esta última passagem é necessária para converter valores em memória para temporários de LLVM.

No capítulo 5, descreveremos a comparação dos resultados para as três estratégias implementadas: apenas funções externas; geração de IR para instruções complexas; e a geração acrescentada pela etapa extra de otimização.

4.3 Representação de Valores Lua Durante a Compilação

Valores em Lua são manipulados diretamente pelas instruções da máquina virtual. Como LLL compila as instruções para IR, a estrutura que representa um valor precisa de mais um nível de abstração durante a compilação. Essa abstração é desempenhada por uma modelagem em classes. Cada valor é tratado por LLL de acordo com sua forma de acesso. Constantes, por exemplo, são conhecidas em tempo de compilação e podem ser otimizadas. Já registradores e *upvalues* deverão ser acessados de diferentes formas durante a execução.

A figura 4.2 apresenta o diagrama de classes deste modelo. A interface *Value* provê métodos abstratos para o acesso ao rótulo e ao valor em si. A implementação da classe *Constant* se beneficia de seus campos serem conhecidos em tempo de compilação. Dessa forma, seus métodos retornam diretamente a

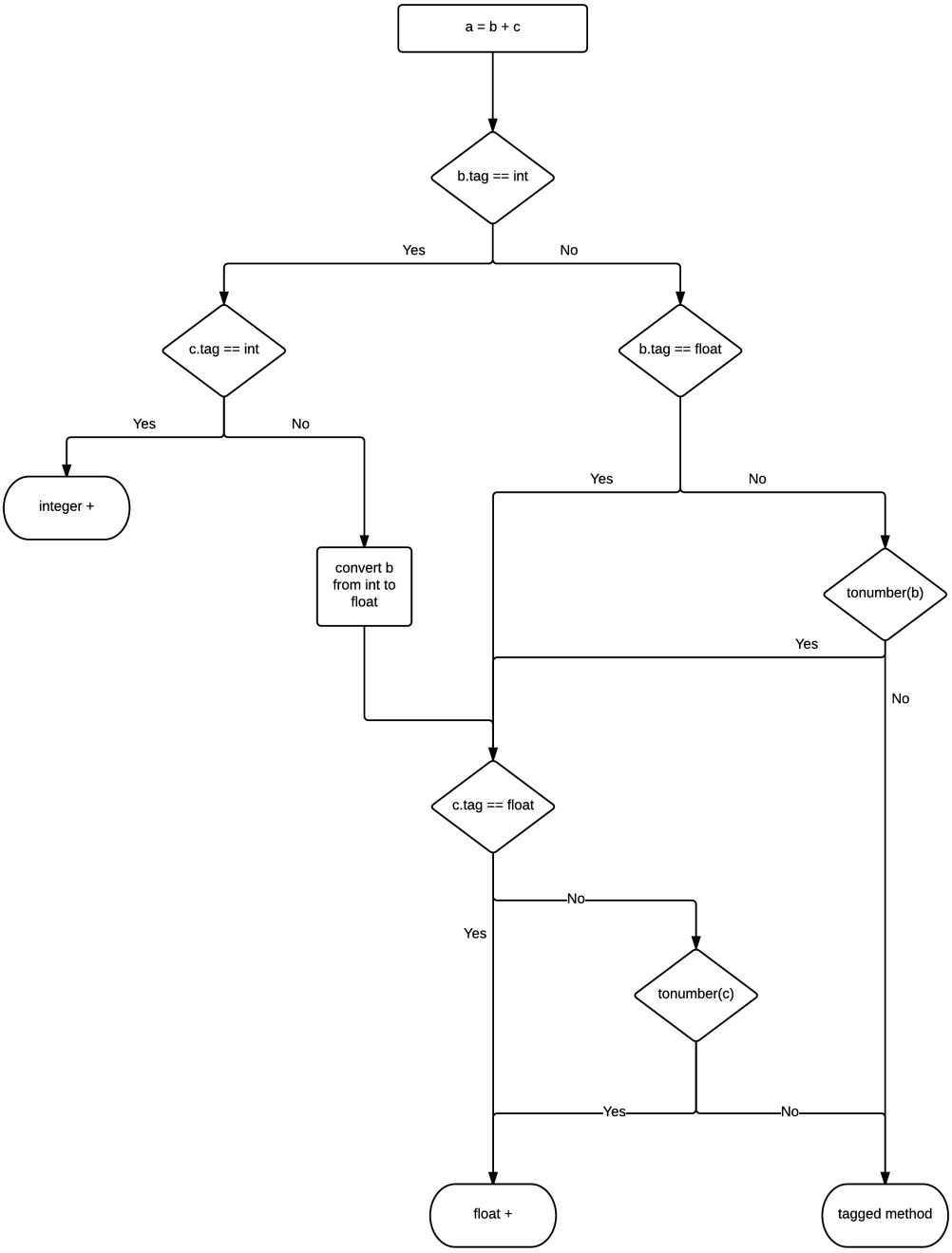

Figura 4.1: Implementação da Operação de Soma em LLL.

constante na representação intermediária.

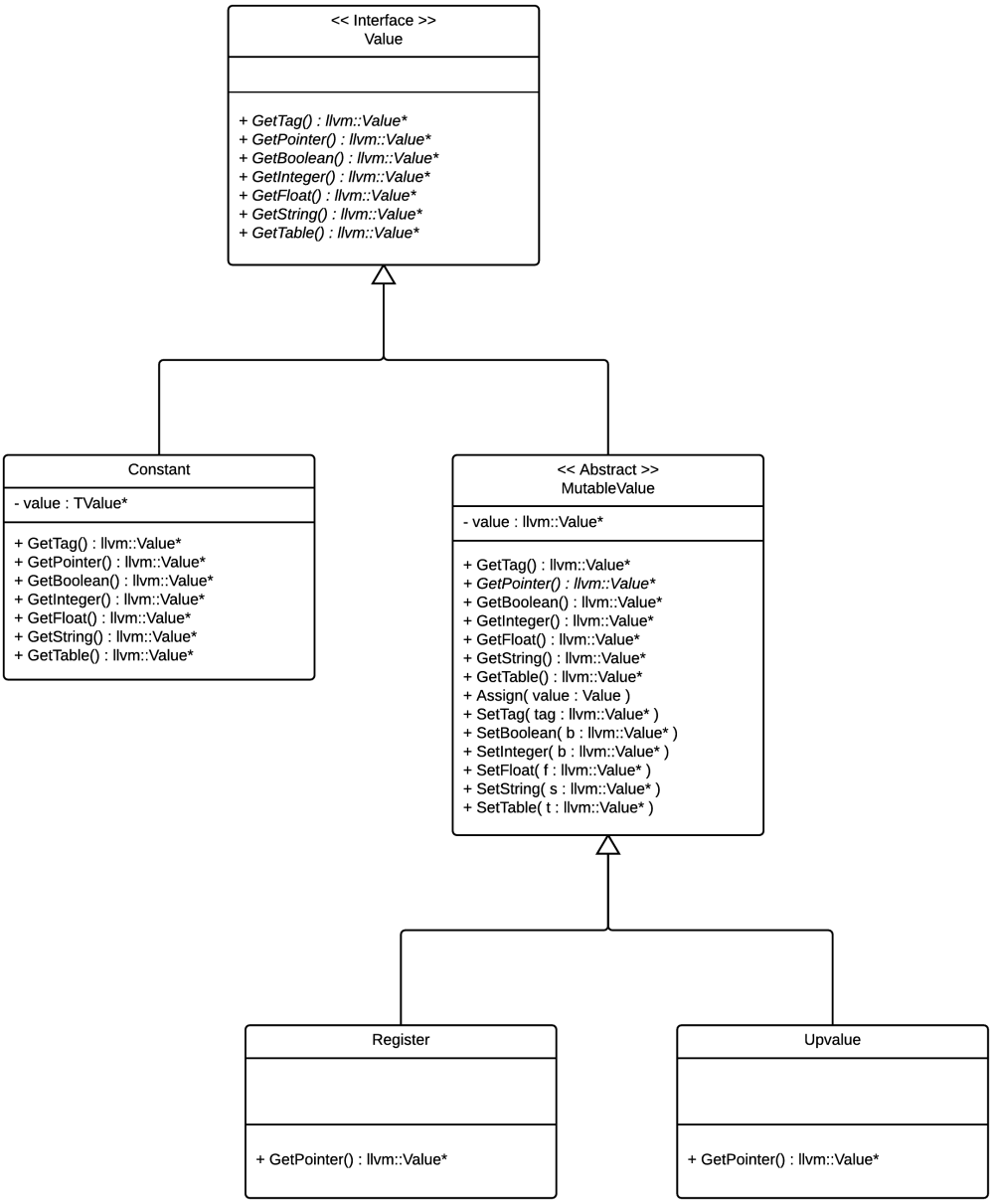

Figura 4.2: Modelo de Classes da Representação de Valores em LLL.

A classe *MutableValue* é abstrata e implementa os métodos comuns entre os valores mutáveis. Nesse métodos, os campos não são conhecidos durante a compilação. Assim, os métodos consistem em gerar o código que irá obter os valores durante a execução. O retorno do método será uma referência para o temporário de LLVM que possui o campo desejado.

No caso de registradores, por exemplo, o valor deverá ser encontrado na

pilha durante a execução. Então, a instrução GEP será utilizada para obter o campo desejado, valor em si ou rótulo. Já no caso de *upvalues*, o valor deverá ser obtido na tabela armazenada no *closure*. Logo, são necessárias duas classes diferentes para tratar o acesso específico à pilha ou à tabela de *closures*.

O polimorfismo provido pela orientação a objetos facilitou consideravelmente o desenvolvimento. Muitas instruções da máquina virtual de Lua tratam registradores e constantes de forma intercambiável. E é essencial que a geração do acesso a essas duas formas de valores seja diferente. Além disso, utilizar diretamente o valor das constantes permite que mais otimizações sejam feitas automaticamente pela LLVM.

Outro desafio encontrado foi tratar a realocação da pilha de registradores. Chamadas de funções fazem com que a pilha cresça, causando a realocação do vetor que a representa. A cada realocação, a variável que aponta para a base da pilha deverá ser atualizada para a nova localização. Com a finalidade de tornar essa atualização simples, utilizamos um espaço de memória dado pela instrução `alloca`. Quando acontecer uma realocação da pilha, a base da pilha será atualizada. Após a geração da representação intermediária, a passagem `mem2reg` converte o espaço em memória para valores temporários. Essa passagem se encarrega de criar as instruções *phi* necessárias para a forma SSA.

Valores de tipos não básicos, como cadeia de caracteres, tabelas e corrotinas, não precisaram de nenhuma representação a mais. A manipulação destes valores é feita por módulos específicos, bastando apenas fazer as chamadas correspondentes.

4.4 Testes e Depuração

No decorrer da implementação de LLL, criamos também testes automáticos para assegurar a corretude do compilador. Como cada instrução é compilada de forma independente, os testes focam na corretude de cada instrução separadamente. Essa abordagem é diferente dos testes já existentes para o interpretador de Lua. A criação de um novo conjunto de testes facilitou consideravelmente a descoberta de erros em LLL.

Todos os testes foram implementados em Lua e divididos em diferentes módulos. Criamos um módulo específico para cada tipo de instrução. Algumas instruções, como as aritméticas, foram agrupadas em um único módulo, pois seus testes são parametrizáveis.

Utilizamos o próprio interpretador de Lua como oráculo para os testes. Para avaliar se uma instrução está correta, primeiro a inserimos numa pe-

quena função. Então, utilizamos a máquina virtual para executar essa função e armazenamos o resultado. Por fim, compilamos a função, a executamos e comparamos os resultados.

Entretanto, inserir as instruções em funções requer que determinadas instruções estejam garantidamente corretas. Essas instruções são as que sempre estão presente em funções, como a MOVE e a RETURN. Esse problema foi contornado ao definir a ordem que os testes são executados. Logo, executamos primeiro os testes de instruções simples e só então testamos as mais complexas.

Para parametrizar os testes, criamos as funções com cadeias de caracteres e então as carregamos em Lua por meio da função `load`. Essa abordagem permitiu a geração automática de diversos casos de teste, especialmente para módulos com muitas instruções similares. O teste das instruções aritméticas, por exemplo, consistiu em todas as combinações dos operadores e de um conjunto variado de argumentos.

4.5 Interface com o Usuário

Não alterar a especificação de Lua foi outros dos principais requisitos da implementação deste projeto. Não inserimos nenhuma nova construção na linguagem, como, por exemplo, tipagem estática. Para controlar o comportamento do compilador JIT, criamos uma biblioteca que é inicializada junto às padrões. A documentação dessa biblioteca está descrita na figura 4.4. Isto posto, existem apenas duas maneiras de compilar uma função: a primeira é manual e a segunda é automática.

A compilação manual é realizada com a chamada da função `lll.compile`. Essa recebe uma função Lua, a compila e retorna um valor booleano indicando se houve sucesso. Vale ressaltar que esse retorno é utilizado apenas durante o desenvolvimento e a depuração, pois todas as funções de Lua devem sempre ser compiladas corretamente. A figura 4.3 exemplifica a utilização dessa funcionalidade.

A compilação automática é ativada por padrão e é realizada quando uma função é chamada determinado número de vezes. O usuário pode habilitar ou desabilitar a compilação automática e definir o número de chamadas necessárias para compilar uma função. O propósito desse modo de funcionamento é livrar o usuário da decisão de quais funções ele deve otimizar, e, portanto, deixar transparente o uso do compilador JIT. Como o foco do projeto era a geração de código de máquina, não implementamos outras heurísticas para identificar *hotspots*.

```
1  -- Declares a Lua function:
2  function foo()
3    -- implementation...
4  end
5
6  -- Executes the function with Lua VM:
7  foo()
8
9  -- Compiles the function:
10 lll.compile(foo)
11
12 -- Executes the dynamic generated code:
13 foo()
```

Figura 4.3: Exemplo de uso da API de LLL.

Função	**Descrição**
`lll.compile(f)`	Compila a função `f` e retorna verdadeiro em caso de sucesso. Caso contrário, a função retorna falso e a mensagem de erro.
`lll.setAutoCompileEnable(b)`	Habilita ou desabilita a compilação automática de acordo com o valor de `b`. A compilação automática é ativada por padrão.
`lll.isAutoCompileEnable()`	Retorna um booleano indicado se a compilação automática está ativada.
`lll.setCallsToCompile(calls)`	Define o número de chamadas necessárias para uma função ser compilada automaticamente. Por padrão esse número é 50.
`lll.getCallsToCompile()`	Obtém o número de chamadas necessárias para uma função ser compilada automaticamente.
`lll.isCompiled(f)`	Retorna um booleano indicando se a função `f` foi compilada.
`lll.debug(f)`	Imprime na tela a representação intermediária de LLVM da função `f` (apenas para depuração).

Figura 4.4: Documentação da interface externa de LLL.

5
Testes de Desempenho

Com intuito de avaliar o desempenho de LLL, realizamos testes medindo o tempo de execução. Testamos diferentes versões de LLL e comparamos os resultados com o interpretador de Lua e com LuaJIT. A seção 5.1 apresenta a especificação dos testes que realizamos. A seção 5.2 apresenta os resultados obtidos e as comparações com o interpretador de Lua. A seção 5.3 apresenta a avaliação do tempo gasto com a compilação. Por fim, a seção 5.4 apresenta comparações de LLL com LuaJIT.

5.1
Especificação dos Testes

Testamos pequenos programas que usam diversas funcionalidades de Lua. Esses programas são apresentados na figura 5.1. No interpretador de Lua, a execução de cada programa demora aproximadamente 10 segundos. Nos testes, compilamos cada programa no inicio da execução, quando executados por LLL. O tempo gasto na compilação também é levado em consideração no tempo total.

Programa	**Descrição**
`loopsum`	Incremento de uma variável dentro de um laço *for*.
`matmul`	Multiplicação de duas matrizes $N \times N$.
`queen`	Resolução do problema das oito damas.
`mandelbrot`	Geração de uma imagem do conjunto de Mandelbrot.
`heapsort`	Ordenação de um vetor com algoritmo *heapsort*.
`sudoku`	Solução de diversos jogos de Sudoku.

Figura 5.1: Programas usados nos testes de desempenho.

Apesar do tempo levado pelos programas ser similar, a complexidade de cada um é diferente. A figura 5.2 expõe tal diferença ao listar o número de instruções portáteis em cada programa. O programa `loopsum` possui 8 instruções, sendo que apenas duas são complexas. No outro extremo, o programa `sudoku` possui 395 instruções, com diversas operações aritméticas e laços aninhados.

Realizamos todos os testes no mesmo computador, que possui o processador Intel Core I7-4770 e o sistema operacional GNU/Linux. Em LLL,

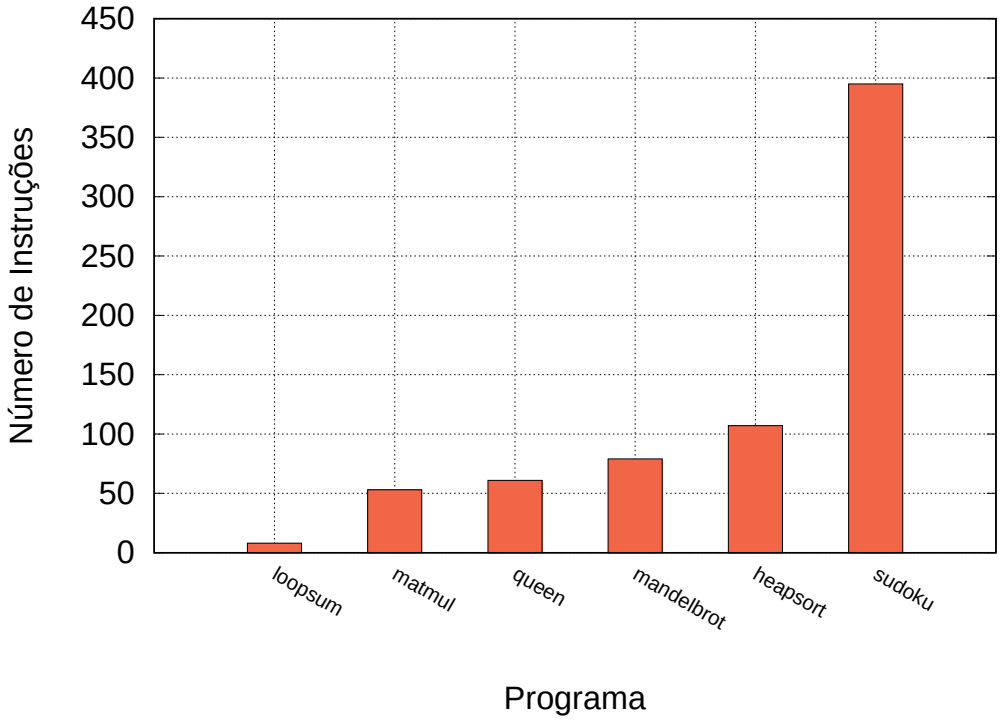

Figura 5.2: Número de instruções portáteis em cada programa de teste.

utilizamos a versão 3.6 de LLVM. Para compilar Lua e LLL, utilizamos a versão 3.6 do compilador Clang, com a flag de otimização `-O3`. Utilizamos a versão 5.3 de Lua para as comparações. Essa foi mesma versão utilizada como base para as modificações feitas em LLL.

Utilizamos um *script* na linguagem Bash para coordenar os testes. O tempo de execução e o uso de memória foram avaliados com a ferramenta `time`. A fim de garantir resultados consistentes, fizemos a média do tempo de 10 execuções. Calculamos também o desvio padrão de cada conjunto de resultados. Pudemos observar os desvios se mantiveram por volta de 1% das médias de tempo.

5.2 Avaliação do Tempo de Execução

Foram avaliadas três diferentes implementações de LLL. A primeira versão compila as instruções portáteis complexas fazendo chamadas a funções externas. A segunda versão gera a representação intermediária para instruções aritméticas e de acesso a tabela. E a terceira versão consiste na segunda versão acrescida de uma etapa de otimização.

O gráfico 5.3 apresenta os resultados obtidos para as diferentes aborda-

gens. Nesse gráfico, o eixo vertical representa a razão do tempo de execução de cada programa em relação ao tempo gasto pelo interpretador de Lua. No programa `loopsum`, por exemplo, a primeira versão de LLL levou 61% do tempo gasto pelo interpretador de Lua. Logo, quanto menor o resultado, melhor o desempenho.

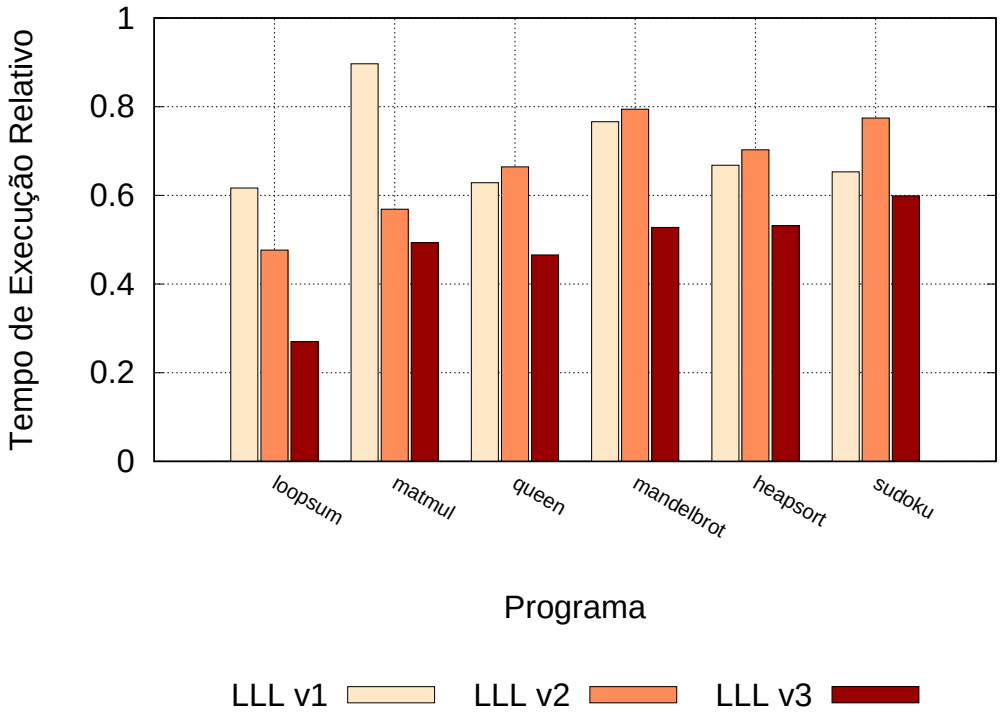

Figura 5.3: Testes de desempenho medindo o tempo de execução. O tempo de execução é relativo à Lua (Lua = 1).

Com base nos resultados, observamos que todas as abordagens foram mais rápidas que o interpretador de Lua. Contudo, no programa `matmul`, o resultado inicial foi pior do que esperávamos. Nesse programa, houve uma redução de apenas 10% do tempo de execução. Na segunda versão, por outro lado, houve uma drástica melhora para esse mesmo programa. Supomos que tal redução ocorreu porque a multiplicação de matrizes consiste basicamente de laços, operações aritméticas e acesso a vetores. Como o código gerado para estas duas últimas instruções foi melhorado na segunda versão, o tempo foi reduzido consideravelmente.

Entretanto, nos outros programas mais complexos, houve um pequeno aumento no tempo gasto. Em especial, no programa `sudoko`, houve um aumento de 12%. Acreditamos que tal perda de desempenho ocorreu por causa da complexidade da representação intermediária gerada.

Finalmente, observamos que o uso de uma etapa extra de otimização permite uma melhora significativa nos resultados. A terceira versão de LLL foi a que obteve o menor tempo em todos os programas. No programa `mandelbrot`, houve uma redução de 27% do tempo de execução. Logo, podemos afirmar que a etapa extra de otimização foi essencial para obter um bom desempenho em LLL.

5.3 Avaliação do Tempo de Compilação

Reduzir o tempo de compilação não foi o foco do projeto, porém ele foi também mesurado. Fizemos comparações entre as três versões de LLL implementadas. O gráfico 5.4 apresenta o tempo relativo gasto na compilação de cada programa. O eixo vertical representa a razão do tempo de compilação em relação à primeira versão de LLL.

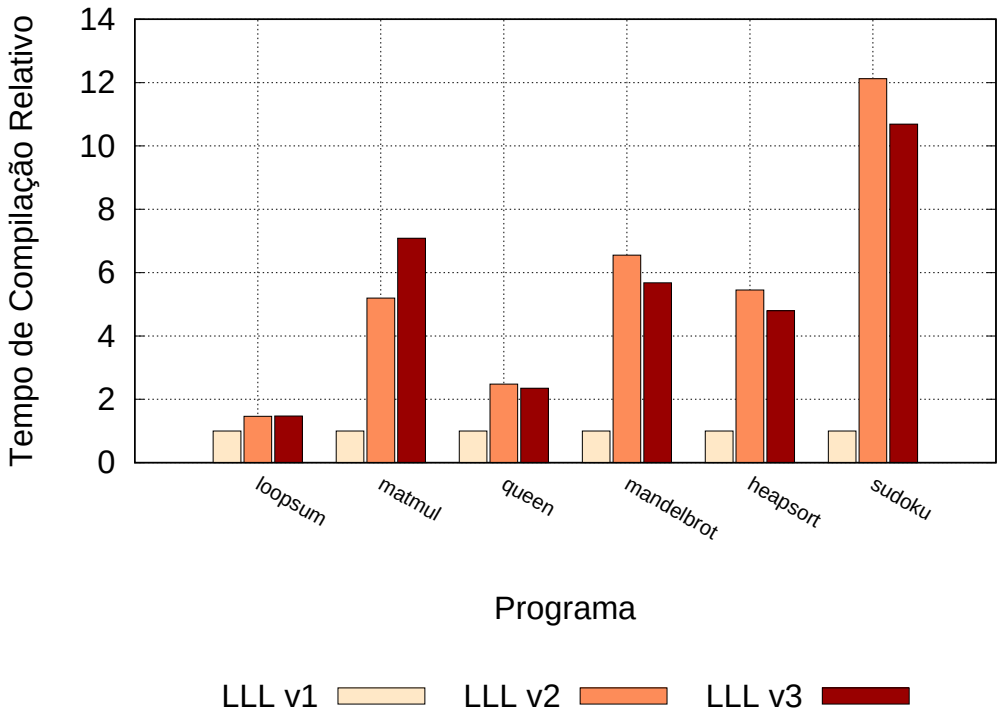

Figura 5.4: Tempo de compilação em LLL para cada abordagem. O tempo é relativo à primeira versão (LLL v1 = 1).

Na primeira versão, a duração da compilação ficava em torno de 30 milissegundos. O programa `sudoku` possuía o maior tempo, que era de 45 milissegundos. Porém, notamos um grande salto no tempo de compilação da primeira

versão para as demais. Na segunda versão, o programa `sudoku` demorou 12 vezes mais tempo para compilar, ultrapassando um segundo.

Contudo, esse comportamento era esperado, pois a geração de IR para as instruções complexas resulta num custo computacional maior nas etapas de otimização. Observamos também que, em alguns casos, o tempo de compilação foi reduzido da segunda para a terceira abordagem. A etapa extra de otimização reduz a complexidade do código passado para as etapas seguintes.

5.4 Comparações com LuaJIT

Por fim, realizamos comparações da solução proposta com LuaJit, o atual estado da arte em termos de desempenho. A figura 5.5 apresenta o tempo de execução de LLL em relação ao de LuaJIT.

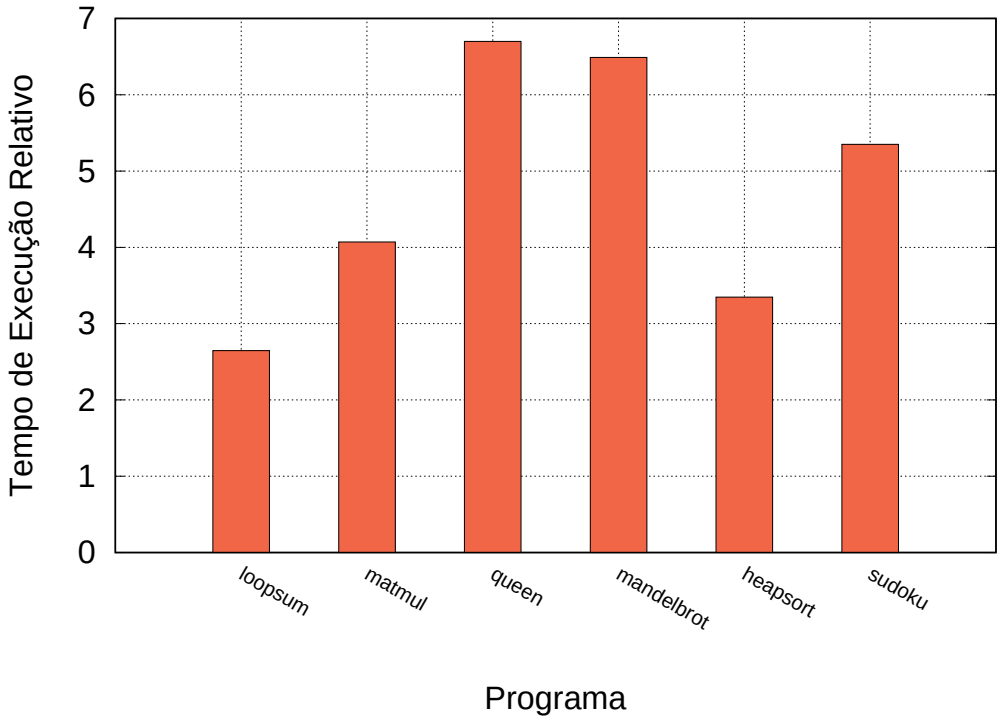

Figura 5.5: Comparação do tempo de execução entre LLL e LuaJit. O tempo de execução é relativo à LuaJIT (LuaJIT = 1).

Observamos que LuaJit obteve um desempenho bem superior em relação a LLL. Para o programa `loopsum`, a execução de LLL foi 2.6 vezes mais lenta que LuaJIT. O pior resultado foi o programa `queen`, que ficou 6.7 vezes mais lento em LLL. Entretanto, vale ressaltar que LuaJit é uma ferramenta mais

madura, que começou a ser desenvolvida em 2005. Ela conta também com outras técnicas de compilação JIT, como *trace-compiling*.

O gráfico 5.6 apresenta o pico de consumo de memória de Lua e LLL em relação ao de LuaJIT. Notamos que Lua gastou mais memória que LuaJIT em quase todos os casos. LLL também gastou mais memória que LuaJIT, chegando até 13 vezes mais. Esse aumento significativo de memória é causado pela LLVM.

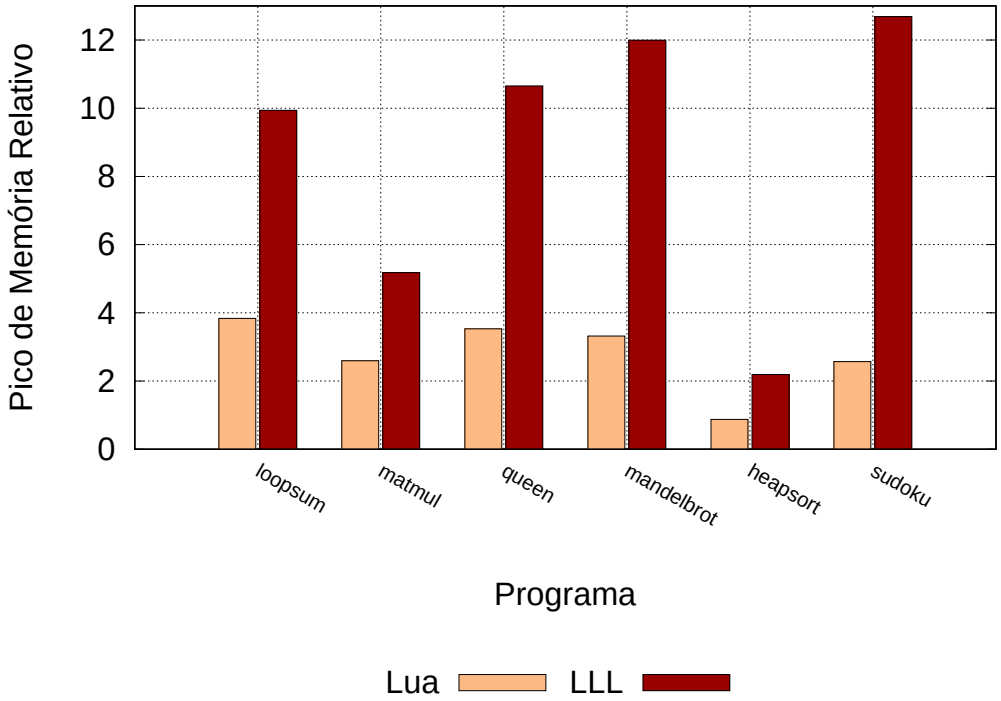

Figura 5.6: Comparação do pico de consumo memória entre Lua, LLL e LuaJit. O pico é relativo à LuaJIT (LuaJIT = 1).

6
Considerações Finais

Este projeto objetivou a implementação de um compilador JIT para Lua utilizando a cadeia de ferramentas LLVM. Para realizar essa implementação, primeiro estudamos a representação intermediária de LLVM. Para exercitar esse estudo, modificamos o compilador de Monga para que ele utilizasse a interface de C de LLVM. Com a modificação do compilador de Monga, pudemos confirmar a viabilidade do uso de LLVM para o projeto.

Após o estudo sobre LLVM, estudamos alguns dos componentes internos de Lua. Esse estudo teve como foco a representação dos valores, as instruções portáteis e a máquina virtual. A medida que estudávamos tais componentes, implementamos um protótipo do compilador JIT proposto. Posteriormente, evoluímos o protótipo de forma que o compilador estivesse completo.

Dada as comparações com o interpretador de Lua, observamos que o compilador que implementamos atingiu um bom resultado em termos de desempenho. Contudo, os resultados obtidos também demonstraram que a abordagem que escolhemos não é tão boa quanto a de LuaJIT. A execução do nosso compilador foi consideravelmente mais lenta que a do atual estado da arte.

Após avaliar os resultados, acreditamos que a melhor abordagem seja de fato a compilação de traços, técnica utilizada por LuaJIT. Nessa técnica, o compilador gera código de máquina para um determinado traço da execução. Um traço é uma sequência de instruções executadas pela máquina virtual. Diferente da compilação de funções, a sequência de instruções só é definida durante a execução do interpretador.

A compilação de traços é consideravelmente mais complexa que a abordagem que escolhemos. São necessárias algumas etapas antes da geração de código de máquina. Primeiro, as instruções JUMP são monitoradas para que *hotspots* possam ser identificados. Quando uma dessas instruções for executada muitas vezes, o compilador passa para a etapa seguinte.

A segunda etapa consiste em gravar as instruções do traço durante a execução da máquina virtual. O traço começa e termina na mesma instrução de JUMP, formando um ciclo. Além da sequência de instruções, também são salvas outras informações que só estão disponíveis durante a execução. As

informações mais importantes a serem salvas são os tipos das variáveis, que, em Lua, só são conhecidos durante a execução.

Após definição do traço, a última etapa trata de gerar o código de máquina. Já que as informações sobre os tipos foram salvas, o compilador consegue gerar um código especializado e eficiente para aquele traço. Em instruções aritméticas, por exemplo, é possível saber qual operação deve ser feita, sem precisar de todas as verificações de tipos.

Entretanto, o código gerado deve também tratar caminhos que não foram executados pela máquina virtual na segunda etapa. No traço, caso haja alguma condição que é sempre verdadeira, deve-se gerar uma instrução de guarda. Essa instrução verificará se a condição é verdadeira e, se não for, a execução do traço será interrompida. Para garantir que o programa funcione corretamente, o compilador deverá recriar o estado da pilha para que o programa possa continuar sendo executado pela máquina virtual.

Pela breve descrição de um compilador de traços, podemos notar que sua implementação não é trivial. Ainda assim, acreditamos que essa seja a melhor forma de criar um compilador JIT de alto desempenho. Em linguagens dinâmicas como Lua, é essencial obter informações em tempo de execução para gerar código de máquina eficiente. Dessa forma, propomos como trabaho futuro a implementação de um JIT para Lua utilizando a técnica de compilação de traços.

A
Exemplo de uso da interface C++ da LLVM

```
/*
 * File:    mcjit.cpp
 * Author:  Gabriel de Quadros Ligneul
 * Date:    Jan, 2016
 *
 * To compile, execute on terminal:
 * clang++ -o mcjit mcjit.cpp `llvm-config --cxxflags --ldflags \
 *                             --libs all --system-libs`
 */

#include <iostream>
#include <memory>

#include <llvm/ADT/StringRef.h>
#include <llvm/ExecutionEngine/ExecutionEngine.h>
#include <llvm/ExecutionEngine/MCJIT.h>
#include <llvm/IR/Constant.h>
#include <llvm/IR/IRBuilder.h>
#include <llvm/IR/LLVMContext.h>
#include <llvm/IR/Module.h>
#include <llvm/IR/Verifier.h>
#include <llvm/Support/DynamicLibrary.h>
#include <llvm/Support/raw_ostream.h>
#include <llvm/Support/TargetSelect.h>

void foo() {
    std::cout << "foo()\n";
}

int main() {
    // Necessary LLVM initializations
    llvm::InitializeNativeTarget();
    llvm::InitializeNativeTargetAsmPrinter();
    llvm::InitializeNativeTargetAsmParser();

    // Create the main module
    auto& context = llvm::getGlobalContext();
    auto module = new llvm::Module("top", context);

    // Declare the external function
    llvm::IRBuilder<> builder(context);
    auto footype = llvm::FunctionType::get(builder.getVoidTy(), false);
    auto foofunc = llvm::Function::Create(footype,
            llvm::Function::ExternalLinkage, "foo", module);
    llvm::sys::DynamicLibrary::AddSymbol("foo", reinterpret_cast<void*>(foo));

    // Define the jitted function
    auto functype = llvm::FunctionType::get(builder.getInt32Ty(), false);
    auto mainfunc = llvm::Function::Create(functype,
            llvm::Function::ExternalLinkage, "main", module);

    // Create the block with the foo's call
    auto entryblock = llvm::BasicBlock::Create(context, "entry", mainfunc);
    builder.SetInsertPoint(entryblock);
    builder.CreateCall(foofunc);
    builder.CreateRet(llvm::ConstantInt::get(builder.getInt32Ty(), 0));

    // Verify the jitted function
    std::string error;
    llvm::raw_string_ostream error_os(error);
    if (llvm::verifyModule(*module, &error_os)) {
        std::cerr << "Module Error: " << error << '\n';
        module->dump();
        return 1;
    }

    // Create the jit engine
    auto engine = llvm::EngineBuilder(module)
                        .setErrorStr(&error)
                        .setOptLevel(llvm::CodeGenOpt::Aggressive)
                        .setEngineKind(llvm::EngineKind::JIT)
                        .setUseMCJIT(true)
                        .create();
    if (!engine) {
        std::cerr << "EE Error: " << error << '\n';
        return 1;
    }

```

```
79      // Compile and execute the jitted function
80      engine->finalizeObject();
81      typedef void (*Function)();
82      Function f = reinterpret_cast<Function>(
83              engine->getPointerToNamedFunction("main"));
84      f();
85
86      return 0;
87  }
```

Referências

[1] R. Ierusalimschy, *Programming in Lua*. Lua. Org, 2013.

[2] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "The implementation of lua 5.0," *J. UCS*, vol. 11, no. 7, pp. 1159–1176, 2005.

[3] J. Aycock, "A brief history of just-in-time," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 97–113, 2003.

[4] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing," in *ACM SIGPLAN Notices*, vol. 48, pp. 105–116, ACM, 2013.

[5] D. Majumdar, "Ravi programming language." http://ravilang.github.io, 2015. Acesso em 29 de junho de 2016.

[6] M. Pall, "The luajit project." http://luajit.org, 2008. Acesso em 29 de junho de 2016.

[7] C. Lattner, *The architecture of open source applications*. CreativeCommons, 2014. Acesso em 29 de junho de 2016.

[8] C. Lattner and V. Adve, *LLVM language reference manual*. Acesso em 29 de junho de 2016.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[10] C. Carruth, "[llvmdev] [rfc] developer policy for llvm c api." http://lists.llvm.org/pipermail/llvm-dev/2015-July/088315.html, 2015. Acesso em 29 de junho de 2016.

[11] *LLVM internal documentation*. Acesso em 29 de junho de 2016.

[12] A. Sen, "Create a working compiler with the llvm framework." http://www.ibm.com/developerworks/library/os-createcompilerllvm1, 2012. Acesso em 29 de junho de 2016.

[13] P. Smith, "How to get started with the llvm c api." https://pauladamsmith.com/blog/2015/01/how-to-get-started-with-llvm-c-api.html, 2015. Acesso em 29 de junho de 2016.

[14] "Bug 20656: Mcjit::addglobalmapping not mapping functions in the current object correctly." https://llvm.org/bugs/show_bug.cgi?id=20656, 2014. Acesso em 29 de junho de 2016.

[15] S. C. Johnson, *Yacc: Yet another compiler-compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[16] M. E. Lesk and E. Schmidt, "Lex: A lexical analyzer generator," 1975.

[17] M. M. Brandis and H. Mössenböck, "Single-pass generation of static single-assignment form for structured languages," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1684–1698, 1994.

[18] K.-H. Man, "A no-frills introduction to lua 5 vm instructions," 2003.

[19] "Performance tips for frontend authors." http://llvm.org/docs/Frontend/PerformanceTips.html. Acesso em 29 de junho de 2016.