Eduardo Mello Cantú

Geração de código para a máquina virtual LLVM a partir de programas escritos na linguagem de programação JAVA (Tradutor Java - LLVM)

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do grau de bacharel em Sistemas de Informação.

Orientador:
Olinto José Varella Furtado

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Florianópolis-SC

2008

Sumário

Lista de Figuras

1 Introdução

A crescente necessidade de código otimizado em diferentes áreas da computação e em particular na área de sistemas embutidos, motivou a criação e o uso da máquina virtual LLVM (Low Level Virtual Machine), para qual, até o momento, só existem front-ends C e C++ e uma versão experimental para Java. Por outro lado a plataforma Java tem se destacado na preferência de diversos segmentos de desenvolvedores, apesar de suas conhecidas limitações de eficiência. Assim sendo, a idéia deste projeto é encontrar uma maneira de permitir que desenvolvedores e pesquisadores que trabalham com a plataforma Java, possam tirar proveito dos benefícios da LLVM.

2 *Objetivos*

O presente trabalho está incluso no contexto de otimização de código, focado nos desenvolvedores que utilizam a linguagem de programação Java. Para isso pretende-se integrar os benefícios da plataforma Java com a eficiência buscada por LLVM.

2.1 Objetivo Geral

O objetivo deste trabalho é estudar detalhadamente a JVM (Java Virtual Machne) e a LLVM para então poder montar um esquema de tradução do bytecode gerado por Java para o código interpretado pela LLVM, buscando integrar os benefícios da plataforma Java com a eficiência da LLVM.

2.2 Objetivos Específicos

Para atingir o objetivo proposto, é necessário realizar alguns passos intermediários antes de partir para o desenvolvimento do esquema de tradução. Desta forma, este trabalho tem como objetivo:

1. Estudar o framework LLVM, juntamente com suas vantagens de utilização
2. Realizar um estudo aprofundado de Java, dando maior ênfase a sua máquina virtual e a estrutura do arquivo interpretado pela mesma, o arquivo class.
3. Propor uma forma de tradução para código da máquina LLVM, a partir de um programa Java compilado em arquivo class.

3 LLVM

Linguagens de programação modernas visam a criação de aplicações mais confiáveis, modulares e dinâmicas , aumentando a produtividade do programador e provendo informações semânticas de alto nível para o compilador. Entretanto, essas características acabam prejudicando a performance da execução de aplicações compiladas.

Situado entre as linguagens de programação modernas e a arquitetura, o compilador é responsável por fazer a aplicação executar da melhor maneira possível. Compiladores fazem isso eliminando processamentos desnecessários e fazendo efetivo uso dos recursos do processador. A solução para os problemas é conceitualmente simples: aumentar o escopo da análise e otimização.

Neste capítulo descrevemos o Low-Level Virtual Machine (LLVM) (5) (6), uma infraestrutura de compilador compatível com as arquiteturas e linguagens de programação modernas, desenvolvida para alcançar três objetivos:

1. Propor uma estratégia agressiva de otimização de múltiplos estágios.
2. Ser um host (hospeiro) de pesquisa e desenvolvimento, provendo uma fundação robusta para projetos atuais e futuros.
3. Operar de forma transparente para o desenvolvedor, comportando-se exatamente como um compilador comum.

O LLVM provê uma excelente performance ao usuário final, um bom tempo de compilação e um ambiente de pesquisa produtivo para desenvolvedores de compiladores.

Um estudo feito com os compiladores existentes, que tinha como intuito avaliar os métodos de produção de executáveis de alta performance, constatou que as técnicas apresentam um alto tempo de compilação.

3.1 Arquitetura do sistema LLVM

A compilação do sistema LLVM é baseada na estratégia de múltiplos passos. Essa estratégia de compilação é única no sentido de permitir uma otimização agressiva ao longo da vida da aplicação.

3.1.1 O design de alto nível de um compilador baseado no LLVM

Comparado aos atuais sistemas de compilação, o sistema LLVM é desenvolvido para efetuar transformações mais sofisticadas no tempo de ligação (do inglês, link-time), execução e após a instalação do software. Com o intuito de ser realisticamente aplicável, o compilador LLVM deve se integrar com esquemas de montagem existentes, e deve ser suficientemente eficiente para ser utilizável em situações corriqueiras.

Na figura 1, vemos o funcionamento geral do sistema LLVM.

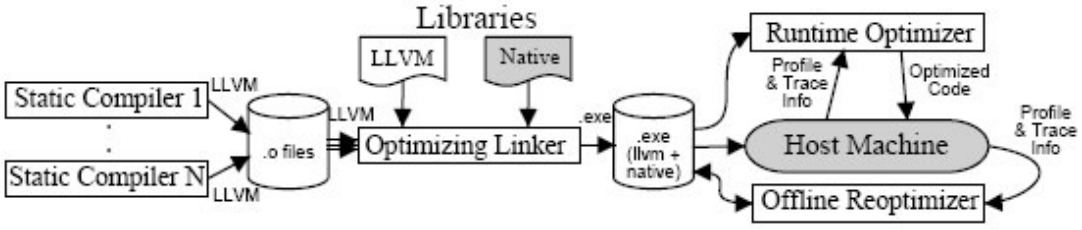

Figura 1: Arquitetura LLVM

Compiladores tradicionais separam o processo de compilação em dois passos: compilação e ligação. Separando em duas fases, teremos os benefícios da compilação separada: apenas as unidades de tradução modificadas devem ser recompiladas. Um compilador tradicional compila o código fonte para um arquivo objeto (.o) contendo código de máquina, e o ligador combina esses arquivos objetos com bibliotecas, para formar o programa executável. Em um sistema simples, tipicamente o ligador faz pouca coisa além de concatenar os arquivos objetos e resolver as referências simbólicas.

A estratégia do LLVM diferencia o tempo de compilação e ligação, aproveitando as vantagens da compilação separada. Ao invés de compilar diretamente para código de máquina, o front-end do compilador estático emite código para um conjunto virtual de instruções do LLVM (do inglês, LLVM virtual instruction set). O ligador-otimizador do LLVM combina os arquivos objeto do LLVM, otimiza-os e então os integra com código nativo executável. Essa organização permite que otimizações interprocedurais sofisticadas sejam executadas no tempo de ligação.

O executável produzido pelo ligador-otimizador contém código de máquina nativo executável diretamente na arquitetura da máquina host, juntamente com um cópia do código LLVM para a própria aplicação. Quando a aplicação é executada, o otimizador de tempo execução monitora a execução do programa coletando informações características que dizem respeito aos padrões de uso da aplicação.

Oportunidades de otimização detectáveis do comportamento da aplicação fazem com que o otimizador de execução re-compile e re-otimize dinamicamente partes da aplicação (usando o código LLVM armazenado). Entretanto, algumas transformações são muito custosas para se efetuar diretamente em tempo de execução. Para essas, o tempo de espera é usado pelo otimizador offline (fora da execução) para recompilar a aplicação usando técnicas interprocedurais agressivas e as informações baseadas nos padrões de utilização do usuário final.

A chave do sistema alto nível LLVM é que o conjunto de instruções virtuais LLVM é usado para comunicação entre diferentes ferramentas, que se encaixam em frameworks de desenvolvimento padrão. Operar sobre uma representação comum, permite que as transformações sejam compartilhadas entre os diferentes componentes do sistema.

3.1.2 Tempo de compilação: Front-end e otimizador estático

O sistema LLVM foi desenvolvido para suportar múltiplos front-ends de linguagens, cada qual traduzindo o código fonte da linguagem suportada para o conjunto de instruções virtuais LLVM. Cada compilador estático efetua, tanto quanto possível, otimizações nas unidades de tradução, para reduzir a quantidade de trabalho do otimizador de tempo de ligação.

O trabalho inicial do front-end de uma linguagem específica é traduzir o código fonte em questão para o conjunto virtual de instruções do LLVM. Adicionalmente, pode também efetuar otimizações específicas da linguagem.

Como todas as transformações do LLVM são modulares e compartilhadas, compiladores estáticos podem escolher a utilização de algumas (ou todas) as transformações da estrutura LLVM para melhorar sua capacidade de geração de código.

Um aspecto importante do conjunto de instruções virtuais do LLVM é a habilidade de suportar código fonte arbitrário, e isso graças a um sistema de tipos de baixo nível. Diferente de máquinas virtuais de alto nível, o sistema de tipos do LLVM não especifica um modelo de objeto, sistema de gerenciamento de memória, ou semânticas específicas de exceções. Ao contrário, o LLVM apenas suporta os tipos de mais baixo nível, como ponteiros, estruturas e

arrays, confiando na linguagem fonte para efetuar o correto mapeamento entre os tipos de mais alto nível.

3.1.3 Tempo de ligação: Ligador e otimizador interprocedural

A ligação é a primeira fase do processo de compilação onde a maior parte do programa fica disponível para análises e transformações e, graças a tal aspecto, o ligador é o local mais indicado para que as análises interprocedurais agressivas.

Finalizado o processo de otimização do ligador, um gerador de código, apropriado para a plataforma desejada, é selecionado para traduzir o código do LLVM para o código da máquina em questão.

3.1.4 Tempo de execução: Profiling e reotimização

Um dos objetivos de pesquisa do projeto LLVM é de desenvolver uma nova estratégia de otimização durante o tempo de execução. Essa estratégia consiste em agrupar informações peculiares durante a execução para usá-las na reotimização e recompilação do programa à partir do bytecode LLVM.

3.1.5 Tempo ocioso: Reotimzador offline

Nem todos os aplicativos são passíveis de otmização em tempo de execução e, com o propósito de suportar essas aplicações, temos o reotimizador offline, que foi desenvolvido para executar durante o tempo ocioso do computador, permitindo otimizações mais agressivas que aquelas realizadas pelo otimizador de tempo de execução.

Ele combina as informações levantadas pelo otimizador do tempo de execução com o bytecode LLVM para reotimizar e recompilar a aplicação.

3.2 Conjunto de instruções virtuais do LLVM (LLVM Virtual Instruction Set, ou LVIS)

Um dos fatores que diferencia o LLVM de outros sistemas é a representação de programa que ele usa. Essa deve ser suficientemente *baixo nível* para permitir a otimização nas fases iniciais da compilação e, suficientemente *alto nivel* para suportar as otimizações agressivas feitas nos tempos de ligação e pós-ligação.

O LVIS , como será chamado de agora em diante, foi desenvolvido como sendo uma representação de *baixo nível* com informações de tipo de *alto nível*.

Ele representa uma arquitetura virtual que captura operações de processadores comuns, mas evita restrições de máquina específicas como registradores físicos, pipelines, chamadas de processos de *baixo nível*, etc. O LLVM permite um conjunto infinito de registadores virtuais capazes de armazenar valores de tipos primitivos. Esses registradores virtuais encontram-se na forma de Static Single Assignement (SSA), que é muito usual nos algoritmos de análise de fluxo de dados.

A maioria das operações do LLVM estão no formato de três endereços, ou seja, com um ou dois operandos geram um resultado. A representação de três endereços é a representação escolhida pela arquitetura RISC. Ela pode ser facilmente comprimida, permitindo uma grande densidade nos arquivos LLVM.

Como o LVIS é a *cola* que une o sistema LLVM, previamente descrito neste capítulo como sendo utilizado na comunicação entre diferentes ferramentas, a eficiência e facilidade de uso do sistema dependem de muitos aspectos do design do conjunto de instruções. Uma importante característica do LVIS é que ele é uma linguagem de primeira classe, com um formato textual, um formato binário comprimido e um formato *em memória* suscetível a transformações.

Abaixo podemos ver a lista das instruções previstas pelo LVIS(2).

- Intruções terminadoras (indicam qual bloco será executado após o fim do bloco atual): *ret, br, switch, invoke, unwind, unreachable*;
- Intruções de operações binárias (são usadas para fazer a maioria das computações em um programa): *add, sub, mul, udiv, sdiv, fdiv, urem, srem, frem*;
- Intruções de operações binarias *bitwise* (para executar várias formas de movimentações de bits): *shl, lshr, ashr, and, or, xor*;
- Intruções de operações vetoriais (utilizadas para acessos de elementos e outras operações específicas de vetores): *extractelement, insertelement, shufflevector*;
- Intruções de operações agregadas (para trabalhar com valores agregados): *extractvalue, insertvalue*;
- Intruções de acesso de memória e operações de endereçamento: *malloc, free, alloca, load, store, getelementptr*;

- Intruções de operações de conversão (usadas para casts): *trunc .. to, zext .. to, sext .. to, fptrunc .. to, fpext .. to, fptoui .. to, fptosi .. to, uitofp .. to, sitofp .. to, ptrtoint .. to, inttoptr .. to, bitcast .. to*;

- Intruções para outros tipos de operações: *icmp, fcmp, vicmp, vfcmp, phi, select, call, va_arg, getresult*.

4 JAVA

A linguagem de programação Java era chamada de Oak. Ela foi desenvolvida por James Gosling para ser embutida em aparelhos eltrodomésticos. Após alguns anos de experiência com a linguagem e grandes contribuições, foi redirecionada para a Internet, renomeada e substancialmente revisada.

Neste capítulo apresentaremos um pouco sobre a história, mas o foco principal será sua máquina virtual (especialmente a estrutura do arquivo *class*), cuja compreensão será de vital importância para este trabalho.

4.1 História

A linguagem de programação Java (4) é uma linguagem de propósito geral, orientada a objetos e concorrente. Sua sintaxe é similar a do C e C++, omitindo os aspectos que às tornam complexa, confusa e insegura. A plataforma Java foi desenvolvida, inicialmente, para solucionar o problema de construir softwares para dispositivos em rede. Para tal propósito, deveria ser compatível com diferentes arquiteturas e permitir a entrega segura dos componentes do software. Para garantir esses requisitos, o código compilado deve *sobreviver* ao transporte através de redes, operar em qualquer cliente e garantir execução segura.

A popularização da WEB tornou esses atributos ainda mais interessantes. A internet demonstrou como conteúdos ricos em mídia eram acessíveis de maneira simples, e a *navegação* na WEB tornou-se popular. Logo concluiu-se que o formato dos documentos HTML para WEB era muito limitado.

O navegador HotJava da Sun demonstrou propriedades interessantes da linguagem e da plataforma Java, tornando possível acoplar *programas* dentro das páginas HTML. Esses *programas* são *baixados* para o navegador HotJava juntamente com a página HTML na qual aparecem. Antes de serem aceitos pelo navegador, os *programas* são cautelosamente verificados para garantir que são seguros. Como as páginas HTML, os *programas* compilados são indepen-

dentes da rede e do provedor. O comportamento dos *programas* são iguais, independentemente de onde vieram ou da máquina para a qual foram carregados.

Um navegador da WEB incorporando a plataforma Java ou Java 2 não é mais limitado a um determinado conjunto de capacidades. Programadores podem escrever o programa uma única vez, e o mesmo irá executar em qualquer máquina que tenha o ambiente de execução (runtime environment) Java ou Java 2.

4.2 A máquina virtual Java (Java Virtual Machine, ou JVM)

A máquina virtual Java (7) é o pilar para a plataforma Java e Java 2. É o componente da tecnologia responsável pela independência do hardware e do sistema operacional, do tamanho reduzido do código compilado e da capacidade de proteger os usuários de programas maliciosos.

A máquina virtual Java é uma máquina computacional abstrata. Como uma máquina real, ela possui um conjunto de instruções e manipula diversas área de memória durante a execução. É extramamente comum implementar uma linguagem de programação utilizando uma máquina virtual e, como exemplo, podemos citar a mais conhecida de todos, a máquina P-Code do UCSD Pascal.

O primeiro protótipo da implementação da máquina virtual do Java, à partir de agora chamado de JVM (do inglês, Java Virtual Machine), desenvolvido pela Sun Microsystems, Inc., emulou o conjunto de instruções da JVM em um software *hospedado* por um dispositivo do tipo *handheld* semelhante ao atual PDA (do inglês, Personal Digital Assistant). As atuais implementações da Sun da JVM, componentes dos produtos JavaTM 2 SDK e JavaTM 2 Runtime Environment, emulam a JVM em *provedores* Win32 e Solaris de maneiras muito mais sofisticadas. Entretanto, a JVM não assume nenhuma implementação de tecnologias particulares, hardwares ou sistemas operacionais *provedores*. Não é interpretada nativamente, mas pode ser interpretada através da compilação do conjunto de instruções de uma CPU, ou implementada em microcódigo ou diretamente no silício.

A JVM não entende nada da linguagem de programação Java, ela só compreende um formato binário particular, o arquivo do formato *class*. O arquivo *class* contém instruções da JVM, conhecidos como bytecode, e uma tabela de símbolos, assim como informações adicionais.

Em nome da segurança, a JVM impõe um formato forte juntamente com amarras estruturais no código em um arquivo *class*. Entretanto, qualquer linguagem com funcionalidades que possam ser expressadas em termos de um arquivo *class* válido pode ser *hospedada* pela JVM.

4.2.1 A estrutura da máquina virtual Java

Antes de adentrar mais profundamente na estrutura da máquina virtual, devemos detalhar a estrutura do arquivo reconhecido pela própria, que é o arquivo *class*.

Compilado para ser executado sobre máquina virtual Java, é um formato binário independente de hardware e sistema operacional, normalmente armazenado em um arquivo conhecido como *.class*. O arquivo *class* define precisamente a representação de uma classe ou interface.

A máquina virtual define várias áreas de dados durante a execução de um programa, como o registrador PC (program counter), sendo que cada thread executada na máquina virtual tem o seu próprio registrador PC; a pilha da máquina virtual (uma para cada thread); o heap, que é compartilhado entre todas as threads da máquina virtual; a área de métodos, que também é compartilhada entre todas as threads; a constant pool do runtime (tempo de execução), que é uma representação da constant pool para cada classe ou interface; e a pilha para métodos nativos, que é uma pilha convencional, coloquialmente chamada de *pilha C*, para armazenar métodos nativos.

Frames são utilizados para armazenar dados e resultados parciais, assim como para efetuar a ligação dinâmica, retornar valores para métodos e lançar exceções.

Um frame é criado cada vez que um método é executado. Os frames são alocadas na pilha da máquina virtual, e cada frame tem seu array de variáveis locais, pilha de operandos e uma referência para a constant pool da classe proprietária do método em questão.

4.2.2 O formato do arquivo *class*

O arquivo *class* contém a definição de uma única classe ou interface. Ele é composto por uma seqüência de bytes de 8 bits.

Para facilitar a explanação, utilizamos estruturas para mostrar o relacionamento entre os elementos formadores do arquivo *class*. Nas estruturas apresentadas desta parte em diante, utilizaremos a seguinte convenção para indicar o seu tamanho: u4 para representar 4 bytes, u3 para 3 bytes, u2 para 2 bytes e u1 para 1 byte.

O arquivo *class* é representado pela estrutura ClassFile, conforme descrito abaixo.

```
ClassFile
{
u4 magic;
u2 minor_version;
u2 major_version;
```

```
u2 constant_pool_count;
cp_info constant_pool[constant_pool_count-1];
u2 access_flags;
u2 this_class;
u2 super_class;
u2 interfaces_count;
u2 interfaces[interfaces_count];
u2 fields_count;
field_info fields[fields_count];
u2 methods_count;
method_info methods[methods_count];
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

Sendo que a definição de cada um dos atributos que formam a estrutura, é a seguinte:

- magic - é número que identifica um arquivo do tipo *class*, que sempre deve conter o valor 0xCAFEBABE;
- minor_version e major_version - formam a versão do arquivo *class*. Se major_version é um número denotado por M, e minor_version denotado por m, a versão do arquivo *class* será M.m;
- constant_pool_count - indica a quantidade de itens presentes no constant pool desta classe;
- constant_pool[] - é um array, que vai da posição 1 até a posição constant_pool_cont - 1, contendo os itens (cp_info) da constant pool;
- access_flags - armazena o valor que representa as permissões de acesso da classe em questão, onde o valor 0x0001 serve para demarcar a classe como public, 0x0010 como final, 0x0020 como super, 0x0200 para caracterizar uma interface e 0x0400 para afirmar que a classe é abstract;
- this_class - é zero ou uma entrada válida no array constant_pool[];
- super_class - é zero ou uma entrada válida no array constant_pool[] que representa a classe que é *mãe* da classe em questão;
- interfaces_count - é o número de superinterfaces diretas desta classe;
- interfaces[] - contém índices válidos do array contant_pool[] que representam as superinterfaces diretas da classe em questão;
- fields_count - é o número de estruturas do tipo field_info contidos dentro do array fields[];

- fields[] - é um array que contém estruturas do tipo field_info;
- method_count - é o número de estruturas do tipo method_info contidos no array methods[];
- methods[] - é um array que contém estruturas do tipo method_info;
- attributes_count - é o número de estruturas do tipo attribute_info contidos no array attributes[];
- attributes[] - é um array que contém estruturas do tipo attribute_info;

Visto os atributos da estrutura principal, podemos agora detalhar cada uma das estruturas que formam esses, que são cp_info, field_info, method_info e attribute_info.

4.2.2.1 Estrutura cp_info

A constant pool é uma tabela composta por um ou mais itens. Cada item é representado pela estrutura cp_info, que tem como propósito armazenar valores referenciados no código.

Abaixo, a representação dessa estrutura.

```
cp_info
{
u1 tag;
u1 info[];
}
```

A definição de cada um dos atributos é a seguinte:

tag - é utilizado para definir qual o tipo da informação que está contida na constant pool; info[] - é o conjunto de informações da entrada, sendo que cada entrada tem sua própria descrição de atributos; Os tipos de entradas da constant pool variam de acordo com o valor do atributo tag da estrutura cp_info, sendo que os mesmos respeitam a seguinte regra (valor da tag e o respectivo tipo de entrada):

- CONSTANT_Class_info

```
CONSTANT_Class_info
{
u1 tag;
u2 name_index;
}
```

Onde:

- tag - conforme descrito na estrutura cp_info, contém o tipo de entrada, neste caso 7, que referencia uma entrada do tipo CONSTANT_Class;
- name_index - é um índice válido da tabela constant_pool[] que, conforme descrito anteriormente, é um item cp_info do tipo CONSTANT_Utf8_info, que brevemente será descrito, representando o nome completo de uma ou interface;

- CONSTANT_Fieldref_info,

  CONSTANT_Methodref_info e CONSTANT_InterfaceMethodref_info

```
CONSTANT_X_info
{
u1 tag;
u2 class_index;
u2 name_and_type_index;
}
```

Onde:

- tag - em CONSTANT_Fieldref_info tem o valor 9, CONSTANT_Methodref tem o valor 10 e CONSTANT_InterfaceMethodref o valor 11;
- class_index - é um índice válido dentro do tabela constant_pool[] que, nessa posição, possui uma estrutura do tipo CONSTANT_Class_info, anteriormente detalhada, que contém a declaração do campo ou método;
- name_and_type_index - índice da constant pool que possui um CONSTANT_NameAndType_info. Se a estrutura for CONSTANT_Fieldref_info representa um nome e descritor de campo. Se for CONSTANT_Methodref_info ou um CONSTANT_InterfaceMethodref_info representa o nome e o descritor de um método.

- CONSTANT_String_info

```
CONSTANT_String_info
{
u1 tag;
u2 string_index;
}
```

Onde:

  - tag - o valor para este tipo específico é 8;
  - string_index - índice válido em constant_pool[] associada a CONSTANT_Utf8_info, representando uma seqüência de caracteres que inicializa o objeto String;

- CONSTANT_Integer_info e CONSTANT_Float_info

```
CONSTANT_X_info
{
u1 tag;
u4 bytes;
}
```

  Novamente, utilizamos a estrutura CONSTANT_X_info, que na prática não existe, para representar estruturas semelhantes, onde:

  - tag - define o tipo da entrada cp_info, com o valor 3 para CONSTANT_Integer_info e 4 para CONSTANT_Float_info;
  - bytes - no caso do tipo CONSTANT_Integer_info, armazena o valor int e, no caso do tipo CONSTANT_Float_info, armazenam o valor float no formato simples do ponto-flutuante IEEE 754;

- CONSTANT_Long_info e CONSTANT_Double_info

```
CONSTANT_X_info
{
u1 tag;
u4 high_bytes;
u4 low_bytes;
}
```

  Onde:

  - tag - valor 5 para CONSTANT_Long_info e 6 para CONSTANT_Double_info;
  - high_bytes e low_bytes - em CONSTANT_Long_info armazena o valor long respeitando a forma ((long) high_bytes ¡¡ 32) + low_bytes e em CONSTANT_Double_info armazena o valor double respeitando o formato duplo do ponto-flutuante IEEE 754.

- CONSTANT_NameAndType_info

```
CONSTANT_NameAndType_info
{
u1 tag;
u2 name_index;
u2 descriptor_index;
}
```

Onde:

- tag - neste caso, com o valor 12;
- name_index - é um índice válido da tabela constant_pool[] associado a uma estrutura do tipo CONSTANT_Utf8_info que representa o nome de um campo ou método válido sendo ou um identificador válido da linguagem Java, ou o nome especial de método ¡init¿;
- descriptor_index - é também um índice válido da tabela constant_pool[] associado a estrutura CONSTANT_Utf8_info, só que esse representa o descritor de uma campo ou método válido.

- CONSTANT_Utf8_info

```
CONSTANT_Utf8_info
{
u1 tag;
u2 length;
u1 bytes[length];
}
```

Onde:

- tag - que neste caso está com o valor 1;
- length - contém o número de bytes dentro do array de bytes;
- bytes[] - é um array de bytes que guarda uma informação no formato de string.

4.2.2.2 Estrutura field_info

Cada campo, vindo da linguagem fonte Java, é descrito por uma estrutura do tipo field_info. Cada item, representado por essa esta estrutura, é armazenado na tabela fileds[]. Em uma classe não pode-se ter dois campos com o mesmo nome e descritor.

Abaixo, a reresentação da estrutura.

```
field_info
{
u2 access_flags;
u2 name_index;
u2 descriptor_index;
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

A definição de cada um dos atributos é a seguinte:

- access_flags - representa as permissões de acesso do campo e pode assumir os valores: 0x0001 para public, 0x0002 para private, 0x0004 para protected, 0x0008 para static, 0x0010 para final, 0x0040 para volatile e 0x0080 para transient;
- name_index - é uma índice válido dentro da tabela constant_pool[] associado a uma estrutura CONSTANT_Utf8_info que representa o nome do campo;
- descriptor_index - é uma índice válido dentro da tabela constant_pool[] associado a uma estrutura CONSTANT_Utf8_info que representa a descrição do campo;
- attributes_count - é o número de atributos contidos no array attributes[];
- attributes[] - contém um lista de estruturas do tipo attribute_info, que serão descrita mais adiante;

4.2.2.3 Estrutura method_info

Cada método derivado de uma classe Java é descrito pela estrutura method_info. Esses são armazenados na tabela methods[]. Uma classe não pode ter mais de um método com o mesmo nome e descritor.

A representação desta estrutura é a seguinte:

```
method_info
{
u2 access_flags;
u2 name_index;
u2 descriptor_index;
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

Onde:

- access_flags - define as permissões de acesso do método, sendo que os valores assumidos são: 0x0001 (public), 0x0002 (private), 0x0004 (protected), 0x0008 (static), 0x0010 (final), 0x0020 (synchronized), 0x0100 (native), 0x0400 (abstract) e 0x0800 (strict);
- name_index - é um índice válido dentro da tabela constant_pool[] da classe em questão, que representa ou os métodos especiais ¡init¿ e ¡clinit¿ ou um método válido da linguagem Java. Esse índice é associado a uma estrutura do tipo CONSTANT_Utf8_info;

- descriptor_index - é um índice válido da tabela constant_pool[] (também associado a uma estrutura CONSTANT_Utf8_info) que representa a descrição do método;
- attributes_count - indica o número de atributos deste método, armazenados no array attributes[];
- attributes[] - é um array de atributos no formato de estruturas do tipo atttribute_info, que será descrita ao longo deste trabalho.

4.2.2.4 Estrutura attribute_info

Atributos são utilizados nas estruturas ClassFile, field_info, method_info (todas essa já detalhadas anteriormente) e Code_attribute, que só será explanada mais a frente.

A representação da estrutura segue o modelo abaixo.

```
attribute_info
{
u2 attribute_name_index;
u4 attribute_length;
u1 info[attribute_length];
}
```

Várias estruturas são na verdade especializações da estrutura básica attribute_info, funcionando mais ou menos com uma herança entre classe. Para todas essas estruturas especializadas, os atributos attribute_name_index e attribute_length têm basicamenete a mesma funcionalidade, variando apenas no propósito da informação neles contida.

Defineremos agora outras estruturas, que como comentando herdam as propriedades da estrutura attribute_info, juntamente com a descrição detalhada dos atributos que as compõem.

- ConstantValue_attribute

  ```
  ConstantValue_attribute
  {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 constantvalue_index;
  }
  ```

  Onde:

  - attribute_name_index - índice válido associado a CONSTANT_Utf8_info no constant pool representando a string *ConstantValue*;

  - attribute_length - deve ser 2, neste caso;
  - constantvalue_index - índice válido em constant pool representando um dos tipo long (CONSTANT_Long), float (CONSTANT_Float), double (CONSTANT_Double), int/short/char/byte/boolean (CONSTANT_Integer) ou String (CONSTANT_String).

- Code_attribute

```
Code_attribute
{
u2 attribute_name_index;
u4 attribute_length;
u2 max_stack;
u2 max_locals;
u4 code_length;
u1 code[code_length];
u2 exception_table_length;
{
u2 start_pc;
u2 end_pc;
u2  handler_pc;
u2  catch_type;
}
exception_table[exception_table_length];
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

  Onde:

  - attribute_name_index - índice válido associado a um CONSTANT_Utf8_info da tabela constant_pool[] represetando a string *Code*;
  - attribute_length - indica o tamanho do atributo;
  - max_stack - indica o tamanho máximo da pilha de operações deste método durante a execução do mesmo;
  - max_locals - indica o número máximo de variáveis alocadas no array variavéis locais;
  - code_length - indica o tamanho do array de bytes (code[]);
  - code[] - é o array de bytes contedo os códigos de operações reais da máquina virtual;
  - exception_table_length - entradas da tabela de exceções (exception_table[]);
  - exception_table[] - é uma tabela contendo as entradas do tipo exception_table, que mais. Cada entrada representa um tratador de exceções do código. Elas são formadas pelos atributos start_pc e end_pc - marcam dentro do array de código (code[])

as faixas onde a exceção é ativada; handler_pc que indica o início do tratamento da exceção; e catch_type, que quando houver, deve ser um índice válido no array constant_pool[] para representar uma classe de exceção;

  - attributes_count - indica o número de atributos contidos na lista de attributos internos (attributes[], que foram descritos previamente);
  - attributes[] - é o array contendo estruturas do tipo attribute_info, que anteriormente foram descritas;

- Exceptions_attribute

```
Exceptions_attribute
{
u2 attribute_name_index;
u4 attribute_length;
u2 number_of_exceptions;
u2 exception_index_table[number_of_exceptions];
}
```

Onde:

  - attribute_name_index - item associado a estrutura CONSTANT_Utf8_info na tabela constant_pool[] que contém a string *Exceptions*;
  - attribute_length - indica o tamanho do atributo;
  - number_of_exceptions - indica a quantidade de itens do array exception_index_table[];
  - exception_index_table[] - contém diversos itens do tipo exception_index_table, sendo que cada um desses é um índice válido dentro do constant_pool[] representado uma classe que é declarada para capturar as exceções lançadas por este método;

- InnerClasses_attribute

```
InnerClasses_attribute
{
u2 attribute_name_index;
u4 attribute_length;
u2 number_of_classes;
{
u2 inner_class_info_index;
u2 outer_class_info_index;
u2 inner_name_index;
u2 inner_class_access_flags;
}
classes[number_of_classes];
}
```

Onde:

- attribute_name_index - índice de constant pool associado a CONSTANT_Utf8_info contendo a string *InnerClasses*;
- attribute_length - informa o tamanho do atributo;
- number_of_classes - indica o número de entradas em classes[];
- classes[] - classes internas a classe proprietária. Os itens do tipo classes possuem os atributos inner_class_info_index, que caso exista, deve ser um item válido em constant pool representando uma classe; outer_class_info_index, que caso exista, deve ser um item válido em constant pool representando uma classe; inner_name_index, se existir, deve ser um item válido em constant pool representando o nome original da classe; inner_class_access_flags, que marcam as permissões da classe, sendo os possíveis valores 0x0001 (public), 0x0002 (private), 0x0004 (protected), 0x0008 (static), 0x0010 (final), 0x0200 (interface) e 0x0400 (abstract);

- SourceFile_attribute

```
SourceFile_attribute
{
u2 attribute_name_index;
u4 attribute_length;
u2 sourcefile_index;
}
```

Onde:

- attribute_name_index - index é um índice em constant_pool[], associado a uma estrutura CONSTANT_Utf8_info, que contém a string *SourceFile*;
- attribute_length - indica o tamanho do atributo, neste caso 2;
- sourcefile_index - índice válido ligado a um CONSTANT_Utf8_info em constant pool representando uma string;

- LineNumberTable_attribute

```
LineNumberTable_attribute
{
u2 attribute_name_index;
u4 attribute_length;
u2 line_number_table_length;
{
u2 start_pc;
u2 line_number;
```

```
}
line_number_table[line_number_table_length];
}
```

Onde:

- attribute_name_index - é um item válido da tabela constant_pool[] que contém uma estrutura do tipo CONSTANT_Utf8_info com a string *LineNumberTable*;
- attribute_length - indica o tamanho do atributo;
- line_number_table_length - indica o tamanho do array line_number_table[];
- line_number_table[] - contém itens do tipo line_number_table com o número da linha no código original indicando uma posição do array code[]. São formados pelos atributos start_pc;
- line_number - contém o número da linha do código original;

- LocalVariableTable_attribute

```
LocalVariableTable_attribute
{
u2 attribute_name_index;
u4 attribute_length;
u2 local_variable_table_length;
{
u2 start_pc;
u2 length;
u2 name_index;
u2 descriptor_index;
u2 index;
}
local_variable_table[local_variable_table_length];
}
```

Onde:

- attribute_name_index - é um item válido da tabela constant_pool[], associado a uma estrutura do tipo CONSTANT_Utf8_info, que contém a string *LocalVariableTable*;
- attribute_length indica - o tamanho do atributo;
- local_variable_table_length - indica o número de itens dentro de local_variable_table[];
- local_variable_table[] - contém os itens do tipo local_variable_table, que representam uma faixa dentro do array code[] que possui o valor de uma variável local. Cada item é composto pelos atributos start_pc, que juntamente com length, indica

a faixa dentro do array code[] onde se encontra a variável; name_index indica um índice válido dentro de constant_pool[] que representa o nome de uma variável local; descriptor_index indica o índice da constant_pool[] que contém a representação da descrição do campo; e index, que indica a posição dentro do array de variáveis locais do frame atual que se encontra variável;

- Deprecated_attribute

```
Deprecated_attribute
{
u2 attribute_name_index;
u4 attribute_length;
}
```

Onde:

  - attribute_name_index - é um item válido da tabela constant_pool[] que contém a string *Deprecated*, está presente em uma estrutura do tipo CONSTANT_Utf8_info;
  - attribute_length - neste caso, tem o valor 0 (zero);

Detalhada a estrutura do arquivo *class*, considerado o pilar da máquina virtual, podemos agora entender melhor como são feitos os processos de carga, ligação e inicialização.

4.2.3 O processo de carga, ligação e inicialização

A máquina virtual Java automaticamente carrega, liga e inicializa classes e interfaces. Carregar é o processo de encontrar uma representação binária de uma classe ou interface com um nome particular e criá-la com essa representação. Ligação é o processo de combinar uma classe ou interface com o estado de execução de uma máquina virtual Java para que assim seja executada. Inicialização de uma classe ou interface consiste em executar seu método de inicialização ¡clinit¿.

A máquina virtual começa o processo criando uma classe inicial. Em seguida ela liga a classe inicial, inicializa-a e invoca o método público void main(String[]). A invocação deste método leva a toda execução subseqüênte.

A criação de um classe ou interface *C* denotada pelo nome *N* consiste na construção, na área de métodos da máquina virtual, de uma implementação específica da representação interna de *C*. A criação de uma classe ou interface é ativada por outra classe ou interface *D*, que referencia *C* através de sua constant pool de execução.

Se *C* não é uma classe do tipo array, ela é criada através da carga da representação binária de *C* usando um class loader (carregador de classes), que pode ser o Bootstrap class loader, ou um class loader definido pelo usuário, devendo esse ser uma subclasse de ClassLoader.

Carregar uma classe através do Bootstrap class loader consiste nos seguintes passos:

1. A máquina virtual procura por uma representação de *C* de uma maneira independente de plataforma. Nessa fase, deve-se detectar o erro NoClassDefFoundError caso nenhuma representação de *C* seja encontrada. Na seqüência a máquina virtual tenta derivar uma classe denotada por *N* usando o Bootstrap class loader, através da representação encontrada, usando o algorítmo específico.

2. O Bootstrap class loader pode delegar a carga de *C* para algum class loader definido pelo usuário, como *L*, por exemplo, passando *N* para uma invocação do método loadClass em *L*. O resultado é a invocação de *C*. Então a máquina virtual armazena a informação de que o Bootstrap class loader é o carregador inicial de *C*.

Agora, para carregar uma classe ou interface atráves de um class loader definido pelo usuário, segue-se os seguintes passos:

1. O class loader *L* pode criar um array de bytes representando *C* como uma estrutura ClassFile; ele deve então invocar o método defineClass da classe ClassLoader. Invocá-lo faz com que a máquina virtual derive uma classe ou interface denotada por *N* usando *L* do array de bytes através do algorítimo específico.

2. O class loader *L* pode delegar a carga de *C* para outro class loader *L'*. Isso é feito passando *N* como argumento, diretamente ou indiretamente, para um método de invocação de *L'*, tipicamente loadClass. O resultado é a invocação de *C*.

O próximo passo para a máquina virtual é fazer a ligação (linking). A ligação (linking) de uma classe ou interface envolve fazer a verificação e preparação da classe ou interface, suas superclasses diretas, superinterfaces diretas e seus elementos, caso necessário. A resolução das referências simbólicas da classe ou interface é uma parte opcional da ligação.

A representação de uma classe ou interface é verificada para garantir que a representação binária é estruturalmente válida. A verificação pode fazer com que classes e interfaces adicionais sejam carregadas.

Uma classe ou interface deve ser verificada com sucesso antes de ser inicializada.

Em seguida, faz-se a preparação. A preparação envolve a criação dos campos estáticos (statics) da classe ou interface e a inicialização desses campos com valores padrão. A preparação não deve ser confundida com a execução de inicializadores estáticos. Diferentemente desses, a prepração não requer a execução de nenhum código da máquina da virtual.

O processo de determinação dinâmica de valores concretos para referências simbólicas na constant pool de execução é chamado de resolução.

Algumas instruções da máquina virtual (anewarray, checkcast, getfield, getstatic,instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, multianewarray, new, putfield e putstatic) fazem referências simbólicas para a constant pool de execução, e a execução dessas instruções requer a resolução dessas referências.

A inicialização de um classe ou interface consiste em invocar seus inicializadores estáticos e os inicializadores dos campos estáticos declarados na classe.

Uma classe ou intercace só deve ser inicializada se uma das instruções (new, getstatic, putstatic e invokestatic) da máquina virtual referenciarem a classe ou interface ou, se algum método reflectivo for invocado ou, se uma de suas subclasses for inicializada ou ainda, se é designada como a classe inicial pela inicialização da máquina virtual.

4.2.4 A lista de instruções da Máquina Virtual Java

A lista de instruções da máquina virtual Java consiste de um opcode (código de operação) especificando alguma operação a ser efetuada, seguida de zero ou mais operandos encorporando valores para serem operados.

Na tabela 1 podemos ver a lista dos códigos de operações e seus respectivos significados.

00	(0x00)	nop	67	(0x43)	fstore_0	138	(0x8a)	l2d
01	(0x01)	aconst_null	68	(0x44)	fstore_1	139	(0x8b)	f2i
02	(0x02)	iconst_m1	69	(0x45)	fstore_2	140	(0x8c)	f2l
03	(0x03)	iconst_0	70	(0x46)	fstore_3	141	(0x8d)	f2d
04	(0x04)	iconst_1	71	(0x47)	dstore_0	142	(0x8e)	d2i
05	(0x05)	iconst_2	72	(0x48)	dstore_1	143	(0x8f)	d2l
06	(0x06)	iconst_3	73	(0x49)	dstore_2	144	(0x90)	d2f
07	(0x07)	iconst_4	74	(0x4a)	dstore_3	145	(0x91)	i2b
08	(0x08)	iconst_5	75	(0x4b)	astore_0	146	(0x92)	i2c
09	(0x09)	lconst_0	76	(0x4c)	astore_1	147	(0x93)	i2s
10	(0x0a)	lconst_1	77	(0x4d)	astore_2	148	(0x94)	lcmp
11	(0x0b)	fconst_0	78	(0x4e)	astore_3	149	(0x95)	fcmpl
12	(0x0c)	fconst_1	79	(0x4f)	iastore	150	(0x96)	fcmpg
13	(0x0d)	fconst_2	80	(0x50)	lastore	151	(0x97)	dcmpl
14	(0x0e)	dconst_0	81	(0x51)	fastore	152	(0x98)	dcmpg
15	(0x0f)	dconst_1	82	(0x52)	dastore	153	(0x99)	ifeq
16	(0x10)	bipush	83	(0x53)	aastore	154	(0x9a)	ifne
17	(0x11)	sipush	84	(0x54)	bastore	155	(0x9b)	iflt
18	(0x12)	ldc	85	(0x55)	castore	156	(0x9c)	ifge
19	(0x13)	ldc_w	86	(0x56)	sastore	157	(0x9d)	ifgt
20	(0x14)	ldc2_w	87	(0x57)	pop	158	(0x9e)	ifle
21	(0x15)	iload	88	(0x58)	pop2	159	(0x9f)	if_icmpeq
22	(0x16)	lload	089	(0x59)	dup	160	(0xa0)	if_icmpne
23	(0x17)	fload	090	(0x5a)	dup_x1	161	(0xa1)	if_icmplt
24	(0x18)	dload	091	(0x5b)	dup_x2	162	(0xa2)	if_icmpge
25	(0x19)	aload	092	(0x5c)	dup2	163	(0xa3)	if_icmpgt
26	(0x1a)	iload_0	093	(0x5d)	dup2_x1	164	(0xa4)	if_icmple
27	(0x1b)	iload_1	094	(0x5e)	dup2_x2	165	(0xa5)	if_acmpeq
28	(0x1c)	iload_2	095	(0x5f)	swap	166	(0xa6)	if_acmpne
29	(0x1d)	iload_3	096	(0x60)	iadd	167	(0xa7)	goto
30	(0x1e)	lload_0	097	(0x61)	ladd	168	(0xa8)	jsr
31	(0x1f)	lload_1	098	(0x62)	fadd	169	(0xa9)	ret
32	(0x20)	lload_2	099	(0x63)	dadd	170	(0xaa)	tableswitch
33	(0x21)	lload_3	100	(0x64)	isub	171	(0xab)	lookupswitch
34	(0x22)	fload_0	101	(0x65)	lsub	172	(0xac)	ireturn
35	(0x23)	fload_1	102	(0x66)	fsub	173	(0xad)	lreturn
36	(0x24)	fload_2	103	(0x67)	dsub	174	(0xae)	freturn
37	(0x25)	fload_3	104	(0x68)	imul	175	(0xaf)	dreturn
38	(0x26)	dload_0	105	(0x69)	lmul	176	(0xb0)	areturn
39	(0x27)	dload_1	106	(0x6a)	fmul	177	(0xb1)	return
40	(0x28)	dload_2	107	(0x6b)	dmul	178	(0xb2)	getstatic
41	(0x29)	dload_3	108	(0x6c)	idiv	179	(0xb3)	putstatic
42	(0x2a)	aload_0	109	(0x6d)	ldiv	180	(0xb4)	getfield
43	(0x2b)	aload_1	110	(0x6e)	fdiv	181	(0xb5)	putfield
44	(0x2c)	aload_2	111	(0x6f)	ddiv	182	(0xb6)	invokevirtual
45	(0x2d)	aload_3	112	(0x70)	irem	183	(0xb7)	invokespecial
46	(0x2e)	iaload	113	(0x71)	lrem	184	(0xb8)	invokestatic
47	(0x2f)	laload	114	(0x72)	frem	185	(0xb9)	invokeinterface
48	(0x30)	faload	115	(0x73)	drem	186	(0xba)	xxxunusedxxx1
49	(0x31)	daload	116	(0x74)	.....ineg	187	(0xbb)	new
50	(0x32)	aaload	117	(0x75)	lneg	188	(0xbc)	newarray
51	(0x33)	baload	118	(0x76)	fneg	189	(0xbd)	anewarray
52	(0x34)	caload	119	(0x77)	dneg	190	(0xbe)	arraylength
53	(0x35)	saload	120	(0x78)	ishl	191	(0xbf)	athrow
54	(0x36)	istore	121	(0x79)	lshl	192	(0xc0)	checkcast
55	(0x37)	lstore	122	(0x7a)	ishr	193	(0xc1)	instanceof
56	(0x38)	fstore	123	(0x7b)	lshr	194	(0xc2)	monitorenter
57	(0x39)	dstore	124	(0x7c)	iushr	195	(0xc3)	monitorexit
58	(0x3a)	astore	125	(0x7d)	lushr	196	(0xc4)	wide
59	(0x3b)	istore_0	130	(0x82)	ixor	197	(0xc5)	multianewarray
60	(0x3c)	istore_1	131	(0x83)	lxor	198	(0xc6)	ifnull
61	(0x3d)	istore_2	132	(0x84)	iinc	199	(0xc7)	ifnonnull
62	(0x3e)	istore_3	133	(0x85)	i2l	200	(0xc8)	goto_w
63	(0x3f)	lstore_0	134	(0x86)	i2f	201	(0xc9)	jsr_w
64	(0x40)	lstore_1	135	(0x87)	i2d	202	(0xca)	breakpoint
65	(0x41)	lstore_2	136	(0x88)	l2i	254	(0xfe)	impdep1
66	(0x42)	lstore_3	137	(0x89)	l2f	255	(0xff)	impdep2

Tabela 1: Tabela de instruções da JVM

5 *O processo de tradução*

Neste capítulo pretende-se entender a teoria por trás do processo da tradução entre linguagens, para então obter-se o conhecimento necessário para realizar a estruturação de uma estratégia viável de desenvolvimento para o trabalho aqui proposto.

Pensando em melhor organizar as etapas, etruturou-se o capítulo da seguinte forma:

- Uma primeira parte dedicada ao entendimento do processo de tradução. Para tanto outros esforços, com propósitos semelhantes a este trabalho, foram levantados por outras equipes, foram averiguados.
- Na segunda parte pretende-se explanar o esforço feito para que fosse possível ser realizada a leitura do bytecode Java (representado pelo arquivo *class*), tarefa essa de extrema importância para o processo de tradução, pois esse será o código fonte do mesmo.
- E, na terceira e última etapa, tem-se como objetivo a proposta de implementação do tradutor responsável pela transformação do código fonte (o bytecode Java) em código objeto (o código LLVM).

5.1 Tradução

A tradução é definida como uma atividade que abrange a interpretação do significado de um texto em uma língua (o texto fonte) e a produção de um novo texto em outra língua mas que exprima o texto original da forma mais exata possível na língua destino; O texto resultante também se chama tradução.

Na área da computação, um tradutor é um programa que traduz um programa ou algo específico (textos principalmente) em outra linguagem ou em atividades.

Como o trabalho em questão está inserido no contexto da tradução entre duas linguagens computacionais, ateu-se a esforços relacionados.

Encontrou-se muitos trabalhos falando sobre o processo de tradução da linguagem Java para código nativo da máquina alvo através de compiladores C. Entre eles, podemos citar o TOBA (8) , que é um sistema para gerar aplicativos Java standalone eficientes.

Jutamente com o TOBA, tem-se um compilador bytecode Java para C, um coletor de lixo, um pacote de threads e suporte para a API Java. A promessa do TOBA é que os aplicativos gerados por ele rodem entre 1.5 e 10 vezes mais rápido que aplicativos interpretados e compilados pelo compilador Just-in-time. Para obter tal feito, o TOBA traduz o arquivo *class* em código C, para então compilá-lo em código de máquina. Os objetos resultantes são então ligados através do sistema de execução do TOBA para criar os tradicionais arquivos executáveis.

O processo de tradução do TOBA consiste em traduzir cada método encontrado no arquivo *class* em uma função C correspondente, conforme a seqüência de passos a seguir:

1. Lê as instruções do arquivo *class*;
2. Computa o estado da pilha para cada intrução;
3. Verifica instruções que são pontos de entrada de exceções e define labels para elas;
4. Verifica intruções alvos de jumps (saltos condicionais) e define labels para elas;
5. Gera cabeçalho de função C e
6. Gera código C para cada instrução.

Além do TOBA, encontrou-se também trabalhos tratando a tradução do bytecode Java diretamente para código assembly (código de montagem) para X86. Em um dos trabalhos (3), os pesquisadores usaram a estratégia de deixar a criação e manipulação dos objetos para a máquina virtual Java. O problema encontrado foi que o código assembly continuava fazendo chamadas freqüêntes para funções da máquina virtual.

Uma alternativa para o problema seria realizar a criação e a manipulação dos objetos através de código assembly, mas isso acabaria a facilidade de uso das funções da máquina virtual. A saída foi um tradutor off-line que permitisse otimizações precisas. Como a máquina virtual Java é uma máquina de pilha, os pesquisadores utilizaram a pilha nativa da arquitetura X86 para emular a pilha de operandos da máquina virtual. Os objetos são alocados na heap da máquina virtual e, estruturas de dados auxiliares criadas pelo tradutor, juntamente com constantes do constant pool do arquivo *class*, são alocados em um arquivo '.data'.

O processo de tradução consiste basicamente em extrair as informações do arquivo *class* e construir tabelas para calcular endereços e gerar instruções assembly.

Os pesquisadores do trabalho em questão concluiram que o código gerado, mesmo obtendo performance superior ao código interpretado, obteve performance inferior ao código submetido ao compilador Just-in-time.

5.2 Mapeamento do arquivo *class*

Aqui descreve-se o esforço feito para obter-se as informações relevantes provenientes do arquivo *class* que, como citado repetitivamente, é o código fonte do processo de tradução proposto por este trabalho.

A primeira tentativa foi a de desenvolver uma ferramenta experimental, uma vez que a estrutura formadora do arquivo *class* já havia sido estudada demasiadamente.

O propósito principal da criação dessa foi meramente educacional, não esperava-se que o mesmo atendesse todas as etapas previstas pela máquina virtual, que são os processos de criação, carga e ligação.

Foi tendo esse pensamento em mente que buscou-se um biblioteca completa, que tivesse como propósito não apenas montar as estruturas da arquivo *class*, mas também realizar todas as etapas necessárias. Assim optou-se pelo *Byte Code Engineering Library*.

Na seqüência, o detalhemento das experiências aqui descritas.

5.2.1 Ferramenta experimental

Como vimos anteriormente, no capítulo referente ao bytecode do Java, a estrutura principal do arquivo *class* seria o ClassFile e dela partiríamos para todas as outras estruturas.

Depois de estudar atentamente o processo, conclui-se que a melhor maneira de representar essas estruturas seria através do paradigma de orientação a objeto, usando UML para representar o relacionamento entre as estruturas, conforme a figura 2.

Optou-se pela linguagem Java para a confecção do aplicativo em questão pois, não bastando a grande afinidade entre o pesquisador e a linguagem, o arquivo *class* é composto por uma seqüência de bytes de 8 bits, que facilmente são lidos através dos métodos readUnsignedByte, readUnsignedShort e readInt da classe java.io.DataInputStream.

O primeiro passo é criar uma classe do tipo ClassFile com auxílio do método estático parse da classe Bytecode2Jvm que recebe como parâmetro um string com o caminho completo do arquivo *class*. O método parse, internamente, criará um novo objeto do tipo ClassFile e invocará

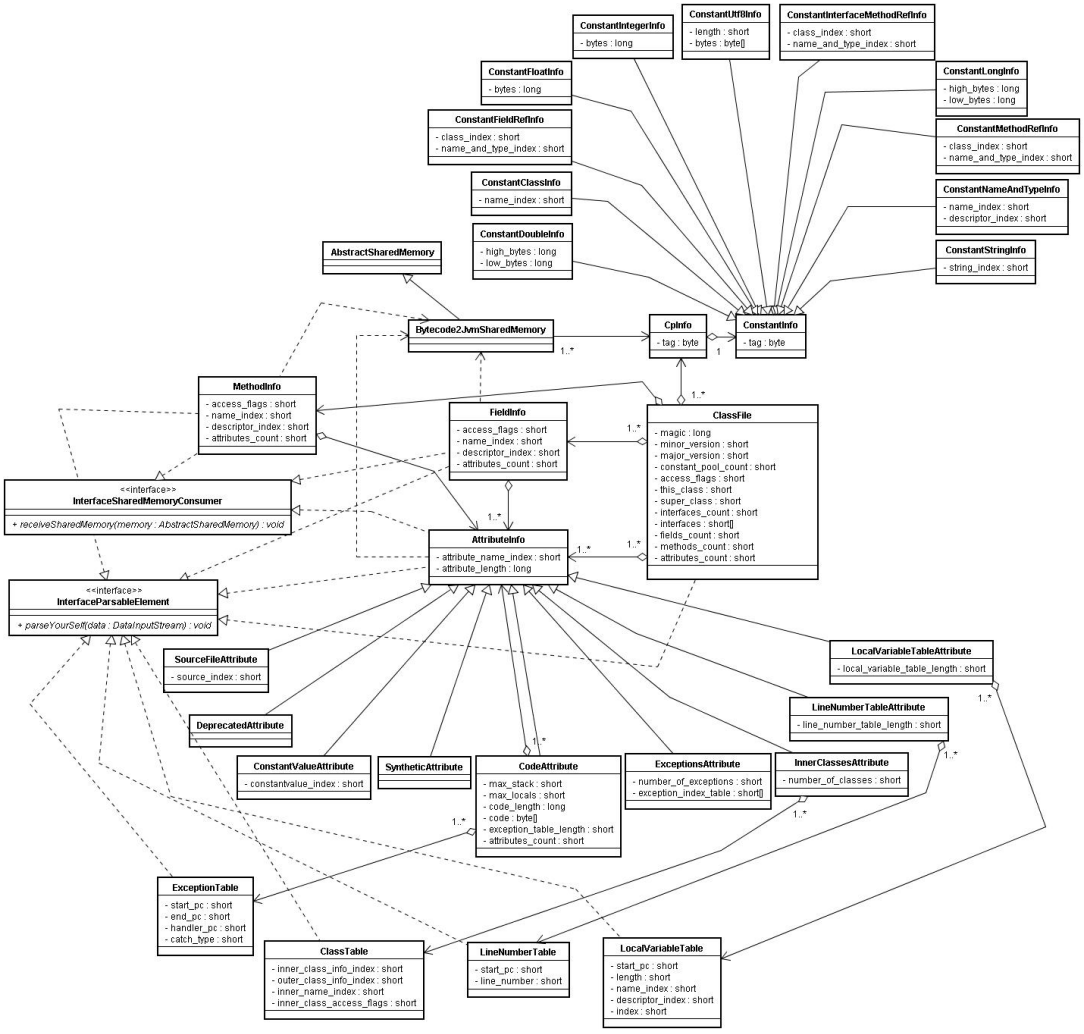

Figura 2: Diagrama de classes

o método parseYourSelf implementado da interface InterfaceParsableElemen, passando como parâmetro o objeto java.io.DataInputStream, criado para ler a seqüência de bytes do arquivo *class*.

Para demonstrar o processo detalhado acima, nada melhor do que um trecho de código explicativo. Para tanto, apresentamos a seguir um exemplo de uso do método responsável pela leitura do arquivo *class*.

```
ClassFile mainClass = Bytecode2Jvm.parse( "C:\\Simples.class" );
System.out.println(mainClass );
```

O arquivo fonte do *class* informado no exemplo acima, Simples.java, segue na seqüência.

```
public class Simples
{
```

```
public static int somar(int x, int y)
{
return x+y;
}
public static void main( String[] args )
{
int x = somar(1,2);
}
}
```

O código do exemplo citado anteriormente faz exatamente o que foi descrito previamente, carrega as estrutras com base no bytecode contido no arquivo *class*.

A seguir podemos ver a saída.

```
magic_number: cafebabe
minor_version: 3
major_version: 45
access_flags: 100001
this_class: 1
super_class: 3
interfaces_count: 0
interfaces[]: [S@186d4c1
fields_count: 0

field info: (INICIO)
field info: (FIM)
methods_count: 3

method info: (INICIO)
index: 19; access_flags: 1001
name_index: 19
descriptor_index: 20
attributes_count: 1

attributes[]: (INICIO)
index: 7; max_stack:2
max_locals:2
code_length:7
code:
      iconst_1
      iconst_2
      invokestatic
      nop
      iload
      istore_1
      return

exception_table_length:0
exception_table[]:
attributes_count:2
index:10; line_number_table_length:2
```

```
line_number_table[]:
start_pc:0
line_number:7

start_pc:6
line_number:8

index:11; local_variable_table_length:2
local_variable_table[]:
start_pc:0
length:7
name_index:23
descriptor_index:24
index:0

start_pc:6
length:1
name_index:16
descriptor_index:17
index:1

attributes[]: (FIM)

index: 5; access_flags: public
name_index: 5
descriptor_index: 6
attributes_count: 1

attributes[]: (INICIO)
index: 7; max_stack:1
max_locals:1
code_length:5
code:
      aload_0
      invokespecial
      nop
      iconst_5
      return

exception_table_length:0
exception_table[]:
attributes_count:2
index:10; line_number_table_length:1
line_number_table[]:
start_pc:0
line_number:1

index:11; local_variable_table_length:1
local_variable_table[]:
start_pc:0
length:5
name_index:12
descriptor_index:13
index:0
```

```
attributes[]: (FIM)

index: 14; access_flags: 1001
name_index: 14
descriptor_index: 15
attributes_count: 1

attributes[]: (INICIO)
index: 7; max_stack:2
max_locals:2
code_length:4
code:
      iload_0
      iload_1
      iadd
      ireturn

exception_table_length:0
exception_table[]:
attributes_count:2
index:10; line_number_table_length:1
line_number_table[]:
start_pc:0
line_number:4

index:11; local_variable_table_length:2
local_variable_table[]:
start_pc:0
length:4
name_index:16
descriptor_index:17
index:0

start_pc:0
length:4
name_index:18
descriptor_index:17
index:1

attributes[]: (FIM)

method info: (FIM)
attributes_count: 1

attributes[]: (INICIO)
index: 25; source_index:26

attributes[]: (FIM)
```

5.2.2 O Byte Code Engineering Library (BCEL)

A API (Application programming interface) do BCEL (1) abstrai a necessidade te conhecer profundamente a máquina virtual Java e como ler e escrever em arquivos binários (*class*). Ela é constituída de três partes:

1. Um pacote que possui classes que descrevem o formato e as estruturas do arquivo *class*. Seu propósito é ler arquivos *class* de um arquivo, ou escrever.
2. Um pacote para dinamicamente gerar ou modificar objetos JavaClass ou Method. Serve para análises de código, remoção de informações necessárias do arquivo *class* ou para implementar um gerador de código.
3. Vários exemplos e códigos.

A figura 3 mostra o relacionamento entre as estruturas que formam o arquivo *class* de acordo com a API BCEL.

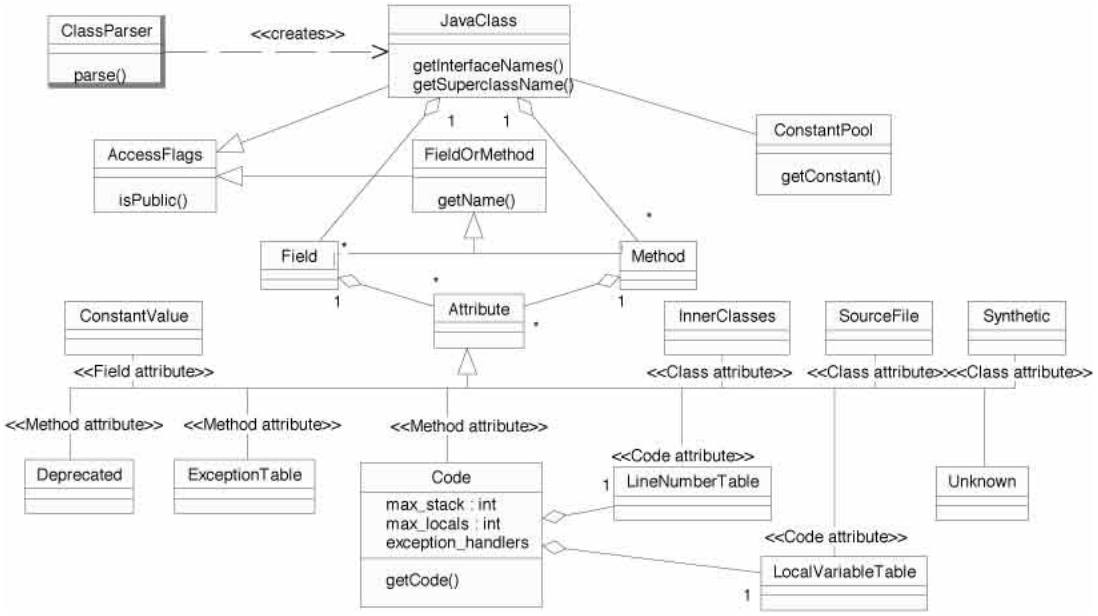

Figura 3: Diagrama de classes

Para enteder melhor o funcionamento da API, fez-se um exemplo, usando como arquivo *class*, o mesmo exemplo anterior, o arquivo Simples.class.

Na seqüência podemos ver o exemplo que monta as estruturas do arquivo *class* utilizando a API BCEL:

```
public static void main(String[] args) throws ClassFormatException, IOException
{
JavaClass mainClass = new ClassParser( "C:\\Simples.class" ).parse();
System.out.println( mainClass );
for ( Method method : mainClass.getMethods() )
{
System.out.println( method );
System.out.println( method.getCode() );
}
}
```

O resultado da execução do exemplo acima é visto abaixo.

```
public class Simples extends java.lang.Object
filename C:\Simples.class
compiled from Simples.java
compiler version 45.3
access flags 33
constant pool 27 entries
ACC_SUPER flag true

Attribute(s):
SourceFile(Simples.java)

3 methods:
public void <init>()
public static int somar(int x, int y)
public static void main(String[] args)

public void <init>()
Code(max_stack = 1, max_locals = 1, code_length = 5)
0:    aload_0
1:    invokespecial java.lang.Object.<init> ()V (8)
4:    return

Attribute(s) =
LineNumber(0, 1)
LocalVariable(start_pc = 0, length = 5, index = 0:Simples this)

public static int somar(int x, int y)
Code(max_stack = 2, max_locals = 2, code_length = 4)
0:    iload_0
1:    iload_1
2:    iadd
3:    ireturn

Attribute(s) =
LineNumber(0, 4)
LocalVariable(start_pc = 0, length = 4, index = 0:int x)
LocalVariable(start_pc = 0, length = 4, index = 1:int y)

public static void main(String[] args)
Code(max_stack = 2, max_locals = 2, code_length = 7)
```

```
0:    iconst_1
1:    iconst_2
2:    invokestatic Simples.somar (II)I (21)
5:    istore_1
6:    return

Attribute(s) =
LineNumber(0, 7), LineNumber(6, 8)
LocalVariable(start_pc = 0, length = 7, index = 0:String[] args)
LocalVariable(start_pc = 6, length = 1, index = 1:int x)
```

Não pode-se deixar de perceber que obteve-se muito mais informações e o manuseio das mesmas é infinitamente mais simples.

Como o propósito do trabalho em questão é propor uma maneira eficiente de traduzir o código fonte, no caso o bytecode Java, para o código objeto, representado pelo LLVM, optou-se pelo uso da biblioteca acima citada para obter-se as intruções da máquina virtual Java.

Referências

1 The byte code engineering library - bcel.

2 Llvm language reference manual.

3 Chun-Yuan Chen, Zong Qiang Chen, and Wuu Yang. Translating java bytecode to x86 assembly code.

4 James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The JavaTM Language Specification Third Edition*. Addison-Wesley, Santa Clara, California, U.S.A., 2005.

5 Chis Lattner and Vikram Adve. A compilation framework for lifelong program analysis and transformation. *UNKNOWN*.

6 Chris Arthur Lattner. LLVM: an infrastructure for multi-stage optimization. Master of science in computer science, Graduate College of the University of Illinois, Urbana, Illinois, 2002.

7 Tim Lindholm and Frank Yellin. *The JavaTM Virtual Machine Specification Second Edition*. Addison-Wesley, 1999.

8 Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications a way ahead of time (wat) compiler. *UNKNOWN*.