Universidade Federal de Uberlândia
Faculdade de Computação

Fábio Moreira Duarte

Transformação de programa em javascript gerado pela ferramenta WebAssembly, para outro de mesma linguagem semanticamente equivalente, com tempo de execução mais eficiênte, utilizando tecnicas de representação computacionalmente mais rapidas por meio do algoritmo GOA

LOCAL
2017

Fábio Moreira Duarte

Transformação de programa em javascript gerado pela ferramenta WebAssembly, para outro de mesma linguagem semanticamente equivalente, com tempo de execução mais eficiênte, utilizando tecnicas de representação computacionalmente mais rapidas por meio do algoritmo GOA

Projeto de pesquisa apresentado como exigência para elaboração do Trabalho de Conclusão de Curso do Bacharelado em Ciência da Computação da Universidade Federal de Uberlândia.

Orientador: Alexsandro Santos Soares

LOCAL
2017

Abreviações

BCC Bacharelado em Ciência da Computação.

FACOM Faculdade de Computação.

GOA Evolutionary optimization of extant software.

M Material.

N NÃO.

NP Não Polinomial.

opcodes código de operação.

P Pergunta.

R Resposta.

S SIM.

UFU Universidade Federal de Uberlândia.

Resumo

Sumário

1 WebAssembly

WebAssembly ou wasm evoluiu do JavaScript, sendo uma linguagem de baixo nivel com formato bytecode para execução no browser.

1.1 Objetivo

Define um binário portátil e eficiênte quanto a espaço e tempo de carregamento, podendo ser compilado para execução em tempo nativo. Seu desempenho de deve ao fato de beneficiar-se das propriedades de hardware disponibilizados pela máquina.

Foi projetado para execução e integração com as plataformas web existêntes Mozzila Firefox e Chrome. Permitindo gerar e executar codigo em velocidade nativa a partir de linguagens C e C++. Utilizando o browser o codigo gerado é executado junto ao JavaScript, atráves de APIs JavaScript próprias WebAssemblyOrg[1].

WebAssembly possui suporte a linguagems em C e C++, para isso foi desenvolvido um backend utilizando Clang/LLVM1.9.1. O que permite o uso de projetos como Emscripten1.9. Abrindo a possibilidade de usuários Emscripten gerarem seus projetos em WebAssembly.

1.2 Ferramenta emsdk

****
EU TENHO QUE FAZER: Explicar o emsdk !!!
****

1.2.1 Instalação

Seu código fonte encontra-se em um repositório no github[2]. É possil obter o fonte a partir da clonagem do conteúdo do repositório, utilizando o comando "git clone", como abaixo.

```
git clone https://github.com/juj/emsdk.git
```

Após o termino do download para a pasta de destino escolhido. Existira um diretório chamado "emsdk". Na pasta criada encontra-se os arquivos do WebAssembly.

É necessário realizar a instalação e atualização para começar a utilizar o emsdk. Para isso deve-se mudar o para o diretório emsdk, utilizando o comando "cd". Como abaixo:

```
cd emsdk
```

Estando no diretório, podemos realizar os seguintes comandos abaixo para iniciarmos a instalação e atualização para a versão mais recente disponível.

```
./emsdk install latest
./emsdk update latest
```

[1] WebAssemblyOrg
[2] http://webassembly.org/getting-started/developers-guide/

Caso o sistema operacional utilizado não possua Emscripten toolchain[3] instalado, o Emscripten SDK[4] pode ser utilizado para direcionar a compilação. Segue abaixo a versão equivalente a de acima.

```
./emsdk install sdk-incoming-64bit binaryen-master-64bit
./emsdk activate sdk-incoming-64bit binaryen-master-64bit
```

Após o termino das etapas acima, a instalação está completa. Sendo possivel compilar um programa em C ou C++ para wasm.

1.2.2 Compilação

Para compilarmos um programa na linguagem C ou C++ para wasm "WebAssembly". Deve-se acessar o ambiente do compilador Emscripten. Segue o comando abaixo:

```
source ./emsdk_env.sh
```

Para demonstração, utilizaremos simples exemplos de código em C 1.2.2. O objetivo é utilizarmos o comando emcc do WebAssembly, para gerarmos o JavaScript e o wasm.

Exemplos:

- Um programa simples, que apenas escreve "Hello, world!" na saida, no caso a saida é o browser escolhido na etapa seguinte.

```c
#include <stdio.h>
int main(int argc, char ** argv) {
{
    printf("Hello, world!\n");
}
```

****

EU TENHO QUE FAZER: ESCLARECER O QUE SÂO OS PARAMETROS DE ENTRADA !!!

****

```
emcc hello.c -s WASM=1 -o hello.html
```

Após a compilação, no diretório estará contido junto com o arquivo original, três novos arquivos como mesmo nome, mas com as seguintes extensões JavaScript, wasm e html.

****

EU TENHO QUE FAZER: PQ ELE GERA UM ARQUIVO JAVASCRIPT e HTML !!!

****

[3] https://kripken.github.io/emscripten-site/docs/building_from_source/toolchain_what_is_needed.html

[4] https://kripken.github.io/emscripten-site/docs/tools_reference/emsdk.html

1.2.3 Execução

Descreve como executar arquivos com extensão .html gerados pelo WebAssembly.

Os comandos abaixo devem ser feitos no mesmo diretório, onde realizou-se o comando abaixo, no guia de instalação 1.2.1.

****

EU TENHO QUE FAZER: O QUE .emsdk_env.sh FAZ !!!

****

```
source ./emsdk_env.sh
```

O comando abaixo lista todos os browsers identificados pelo emrun.

```
emrun --list_browsers
```

Dentre as opções listadas pelo comando acima, optou-se pela utilização do navegador Firefox.

O fonte dos exemplos abaixo, estão em exemplos 1.2.2.

- Execução do hello.html.

  ```
  emrun --browser firefox hello.html
  ```

  Resultado apresentado na figura 1.1.

Figura 1.1 – Printscreen do resultado da execução do hello.html

Mais detalhes e opções estão disponiveis com o comando.

```
emrun --help
```

1.3 Ferramenta WABT

A ferramenta WABT foi criada para permitir a manipulação do wasm utilizando editores de texto expostos, como browsers. Ela permite a conversão de WebAssembly em formato de texto para wasm.

1.3.1 Instalação

Seu código fonte encontra-se em um repositório no github[5]. É possível obte-lo é atravez da clonagem do repositório utilizando o comando "git clone".

No diretório escolhido para manter o código, realize o seguinte comando em um terminal (O comando iniciará o download dos arquivos necessários).

```
git clone --recursive https://github.com/WebAssembly/wabt
```

Após o termino do download, um diretório chamado wabt foi criado. Utilize o comando abaixo, para encaminhar-se para o diretório partindo do diretório atual.

```
cd wabt
```

No diretório wabt estara contido todos os arquivos do repositório. Antes de utilização é necessário construi-los. Os comandos de construção dependem do sistema operacional utilizado pela máquina.

Construtor para Mac e Linux

Dentro do diretório wabt, realize o comando abaixo. Este comando requer a ferramenta CMake[6]. Em sistemas operacionais Mac é necessário uma versão do CMake superior ou igual a versão 3.2 .

O comando construirá a versão default da ferramenta, sendo constituida de um debugador e construtor. O debugador e construtor utilizam-se do compilador Clang.

```
make
```

Construtor para Windows

Dentro do diretório wabt, realize o comando abaixo. Este comando requer o pacote CMake[7], alem disso é necessário a ferramenta Visual Studio[8] ou o compilador MinGW[9]. Caso opte pelo Visual Studio, será necessário o uso de uma versão de 2015 ou superior.

O comando construirá a versão default da ferramenta, um debugador e construtor utilizando o compilador Clang.

[5] https://github.com/WebAssembly/wabt.git
[6] https://cmake.org/
[7] https://cmake.org/
[8] https://www.visualstudio.com/
[9] http://www.mingw.org/

Parâmetros: O parâmetro “config” deve ser um tipo de construtor do CMake, o parâmetro “generator” deve conter o tipo de programa que se deseja gerar.

Utilizando o comando “cmake –help” é possível identificar os tipos de geradores disponíveis.

```
cd [build dir]
cmake [wabt project root] -DCMAKE_BUILD_TYPE=[config] -DCMAKE_INSTALL_PREFIX=[install directory
```

1.3.2 Fim da instalação:

Após a conclusão das etapas acima, a instalação e configuração da ferramenta wabt, estara concluida, sendo possível a geração de um código em wasm a partir de em wast.

Criando um caminho

Após os comandos de construção, o resultado será colocado em “out/clang/Debug/”.

Para facilitar o chamado do metódo, podemos criar um caminho com o comando abaixo (o comando deve ser executado a partir do diretório wabt).

```
ln -s $(pwd)/out/clang/Debug/wat2wasm ~/Documents/WebAssmbler/wabt/out
```

****
EU TENHO QUE FAZER: Rever o comando acima, não parece certo !!!
****

Exemplos

Por motivo de demonstração, utilizaremos o exemplo1.wast[10] abaixo.

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "sumtwo" (func $sumtwo))
  (func $sumtwo (param $0 i32) (param $1 i32) (result i32)
    (i32.add
      (get_local $1)
      (get_local $0)
    )
  )
)
```

O código encontra-se dentro do diretório Examples.

```
out/wat2wasm Examples/example1.wat -o Examples/example1.wasm
```

Após a execução, dentro do diretório do exemplo estará contido o arquivo example1.wasm, que é o equivalente ao example1.wast.

```
out/wat2wasm Examples/example1.wat -o Examples/example1.wasm
```

A formatação do wasté definada na seção 1.6.

[10] https://developer.mozilla.org/en-US/docs/WebAssembly/Text_format_to_wasm

1.4 JavaScript

A partir de um programa em JavaScript é possivel invocar os métodos definidos no binário com extensão wasm. Para isso utiliza-se de métodos de importação e exportação de função.

Primeiro le-mos o contêudo do arquivo wasm, este contéudo deve ser transformado em um array de bytes para instanciação do modulo WebAssembly, seção 1.8. A partir do modulo e de um JSON contendo a quantidade de memoria que será utilizada e a tabela de funções contendo parâmetros de entrada e saida.

1.4.1 Execução utilizando html

Pode-se executar o JavaScript por meio do html em navegadores com suporte ao WebAssembly. Navegadores como FireFox e Chrome citar as versões suportadas, possuem suporte ao WebAssembly. O que possibilita a execução com sucesso do JS com wasm.

Exemplo: O exemplo abaixo 1.4.1, está no formato wat. O algoritmo define a função "sumtwo", ela recebe dois inteiros como parâmetro de entrada e retorna a soma dos dois elementos. Copie o exemplo e salve-o em um arquivo com extensão wat, "sumtwo.wat".

Utilizando a ferramenta wabt seção 1.3, pode-se compilar um arquivo em wat para wasm. Aplicando a ferramenta wabt no exemplo abaixo "sumtwo.wat", teremos o wasm do arquivo.

```
(module
  (table 0 anyfunc)
  (memory $0 1)
  (export "memory" (memory $0))
  (export "sumtwo" (func $sumtwo))
  (func $sumtwo (param $0 i32) (param $1 i32) (result i32)
    (i32.add
      (get_local $1)
      (get_local $0)
    )
  )
)
```

Listing 1.1 – sumtwo.wat

Para utilização do arquivo em formato wasm"sumtwo.wat", pode-se utilizar um arquivo html como 1.4.

No arquivo abaixo, importamos o JavaScript 1.4.1.

```
<!doctype html>

<html>

  <head>
    <meta charset="utf-8">
    <title>WASM test</title>
  </head>

  <body>
```

```
    <script src="./sumtwo.js"></script>
  </body>

</html>
```

Listing 1.2 – example.html

O JavaScript 1.4.1 é responsável por invocar as funções do wasm gerado a partir do programa wat 1.4.1. Copie o contéudo abaixo dentro de um arquivo sumtwo.js, no mesmo diretório do arquivo html 1.4.1.

```
var importObject = {
        imports: {
          imported_func: function(arg) {
            console.log(arg);
          }
        }
      };

fetchAndInstantiate('sumtwo.wasm').then(function(instance) {
   console.log(instance.exports.sumtwo(1, 2));
});

function fetchAndInstantiate(url, importObject) {
  return fetch(url).then(response =>
    response.arrayBuffer()
  ).then(bytes =>
    WebAssembly.instantiate(bytes, importObject)
  ).then(results =>
    results.instance
  );
}
```

Listing 1.3 – sumtwo.js

Tendo o JavaScript “sumtwo.js”, o html “example.html” e o arquivo wasm “sumtwo.wasm”.

Abrindo o arquivo html 1.4.1 em um navegador com suporte ao WebAssembly, o JavaScript 1.4.1 conseguira invocar o método “sumtwo” do wasm 1.4.1 com sucesso. Inspecionando a página html, encontra-se o valor 3 na aba console, correspondente a operação “sumtwo(1,2)” que retorna o valor resultado da soma dos parâmetros.

1.4.2 Execução utilizando NODEJS

É possivel construir um JavaScript equivalente ao JS 1.4.1 mas que seja executado em node como o arquivo JS abaixo.

```
const fs = require( 'fs' ),
    buffer = fs.readFileSync( './add.wasm' ),
    arrayBuffer = new Uint8Array( buffer ).buffer;

var instance;

```

```
WebAssembly.compile( arrayBuffer ).then( module => {
    let imports = {
      env : {
        memoryBase : 0,
        tableBase : 0
      }
    };

    if( !imports.env.memory )
        imports.env.memory = new WebAssembly.Memory({
            initial: 256
        });

    if( !imports.env.table )
        imports.env.table = new WebAssembly.Table({
            initial: 0,
            element: 'anyfunc'
        });

    instance = new WebAssembly.Instance(module, imports);
    console.log(instance.exports.sumtwo(1,2));
});
```

Listing 1.4 – sumtwo.js

Para a execução das funções definidas do arquivo wasm gerado a partir do arquivo wat 1.4.1. Utilizou-se o nodejs versão 8.6.0, seção 1.5.

```
n use 8.6.0 --expose-wasm add.js
```

1.5 Nodejs

Para execução de programas em JavaScript capazes de invocar métodos definidos em um binário wasm, utilizou-se o nodejs na versão 8.6.0 . Versões inferiores podem não suportar o JS.

1.5.1 n

Para instalação de uma versão específica do nodejs, foi utilizado o gerenciador de versões do node[11].

Instalação: para instalação foram utilizados os comandos abaixo:

```
sudo npm cache clean -f
sudo npm install -g n
sudo n 8.6.0
```

[11] https://github.com/tj/n

1.5.2 Execução:

Para compilar a versão do JavaScript 1.4.2, capaz de invocar as funções definidas em um código wasm, utilizou-se os comandos abaixo:

O comando foi executado dentro do diretório MyExamples.

```
n use 8.6.0 --expose-wasm add.js
```

1.6 Linguagem wast

a linguagem wast é uma representação textual do do WebAssembly, que encontra-se em formato binário, ela foi criada para facilitar a leitura e edição.

Tano formato wast quanto no formato wasm o modulo 1.8 é fundamental. Em wast o modulo é representado como uma árvore, sendo o modulo uma árvore como nós onde os nós descrevem sua estruta e código.

****

EU TENHO QUE FAZER: Buscar fontes que comprovem que uma linguagem em arvore é interpretada de maneira mais rapida !!!

****

1.6.1 Conversão de wast para wasm

WebAssemblyutiliza encodificação binária para representar os dados, funções e etc, seção 1.7. Durante o processo de compilação utilizando a ferramenta wabt, seção 1.3, um programa escrito em wast como no exemplo 1.3.2 é convertido na sua representação equivalente em wasm.

Representações

Seguindo as representações da seção 1.7, pode-se relacionar os tipos de representações que equivalem a um dado fragmento de código em wast.

A linguagem wasté do tipo S-expression, onde é representada com formato de árvore.

module **********

EU TENHO QUE FAZER: criar a tabela em binaryEncoding e referenciar aki !!!

****

```
(module)
```

Em wasm, module é representado da seguinte maneira.

```
0000000: 0061 736d                 ; WASM_BINARY_MAGIC
0000004: 0d00 0000                 ; WASM_BINARY_VERSION
```

Funções Em wastas funções seguem o formato abaixo, onde o nó da função recebe três outros nós, o primeiro nó "signature" é responsável por declarar os parâmetros e retornos da função, como no exemplo 1.6.1. Locals são varíaveis que são associadas ao parâmetro como no exemplo 1.6.1

```
( func <signature> <locals> <body> )
```

Parâmetro e retornos da função Define o tipo do parâmetro e o tipo do retorno.

```
(func (param i32) (param i32) (result f64) ... )
```

Locais da função Associa a vaŕiavel como o parâmetro.

```
(func (param i32) (param f32) (local f64)
  get_local 0
  get_local 1
  get_local 2)
```

****

EU TENHO QUE FAZER: Melhorar isso !!!

****

1.7 Encodificação binária

Encodificação binária é uma maneira de representar informação, nela a informação é contida de maneira densa, gerando arquivos menores o que consome menos tempo de decodificação e reduz o espaço necessário em memória.

****

EU TENHO QUE FAZER: Talvez um artigo explicando os benéficios do uso de encodificação binária, seja interessante, quanto a codificação , decodificação e espaço !!!

****

1.7.1 Encodificação do WebAssembly

O WebAssembly utiliza de encodificação binária densa. Sua encodificação divide-se em três camadas [2]:

Camadas

1. Representa instruções e dados relacionados. É representada em bytecodes.
2. Complementa a camada acima, provendo dados sobre a árvore sintática e seus nós.
3. É responsável pela compactação dos dados através do uso de algoritmos de compressão.

1.7.2 Representações

Dados WebAssembly trabalha com três formatos de dados. Inteiros sem sinal de tamanho fixo, inteiro sem sinal de tamanho variável e inteiro com sinal de tamanho varíavel.

uintN é a representação de um inteiro sem sinal, com N numero de bits, sendo representado em N/8 bytes. N pode assumir os valores de 8, 16 ou 32.

varuintN é a representação um inteiro sem sinal, possui tamanho variável, sendo limitado por N bits. N pode assumir os valores de 1, 7 e 32.

varintN é a representação de um inteiro com sinal, limitado a N bits. N pode assumir os valores de 7, 32 e 64.

Instruções WebAssemblypossui menos de 256 instruções Opcodes, o que permite a representação com o uso de 1 byte [2].

Tipos de linguagem Todos os tipos de linguagem são representados por valores negativos da representação varint7 [2], tipos de dados 1.7.2. Os tipos podem ser observados na tabela 1.1.

Tabela 1.1 – Tipos de linguagem, representados por varint7.

Representação opcode	Tipo de construtor
-0x01	i32
-0x02	i64
-0x03	f32
-0x04	f64
-0x10	anyfunc
-0x20	func
-0x40	representação de um bloco vazio, block_type

fonte: Tabela retirada da documentação disponibilizada online, pelo WebAssembly[2]

Tipos de valores

WebAssemblytrabalha com inteiros e pontos flutuantes, cada um possuindo representações de 32 e 64 bits, sendo representados utilizando varint7, tipos de dados 1.7.2.

i32 representa o tipo inteiro de 32 bits

i64 representa o tipo inteiro de 64 bits

f32 representa o tipo ponto flutuante de 32 bits

f64 representa o tipo ponto flutuante de 64 bits

Tipos de funções A assinatura da função e o seu tipo de construtor, está defino na tabela 1.2.

Tabela 1.2 – Assinatura das funções e tipo de construtor.

Campo	Tipo	Descrição
form	varint7	Valor do construtor do tipo da função
param_count	varuint32	Numero de parâmetros
param_types	value_type*	Tipos dos parâmetros
return_count	varuint1	Numero de resultados da função
return_type	value_type?	Tipo da função (se return_count for 1)

fonte: Tabela retirada da documentação disponibilizada online, pelo WebAssembly[2]

****
EU TENHO QUE FAZER: continuar a explicar os tipos !!!
****

1.8 Modulo

****
EU TENHO QUE FAZER: aprofundar no module !!!
****

1.9 Emscripten

****
EU TENHO QUE FAZER: O QUE É EMSCRIPTEN E PQ WEBASSEMBLY USA ISSO !!!
****

Emscripten é baseado em LLVM, compilando C e C++ para um JavaScript otimizado em formato asm. Permitindo a execução de codigo em velocidade nativa, sem a necessidade de plugins.

1.9.1 LLVM

O projeto LLVM é um coleção de compiladores e tecnológias de encadeamento. Com objetivo de prover uma estratégia de compilação baseada em SSA, suportando compilação estática e dinâmica de linguagens de programação arbitrárias [1, 3].

A figura 1.2, descreve o a estratégia adotada. A linguagem fonte, deve ser capaz de gerar uma IR "Representação intermediária".

Sendo gerada a versão intermediária, a partir das configurações disponíveis, o LLVM se encarrega de gerar a versão para a máquina de destino caso a máquina esteja definida dentre as suportadas.

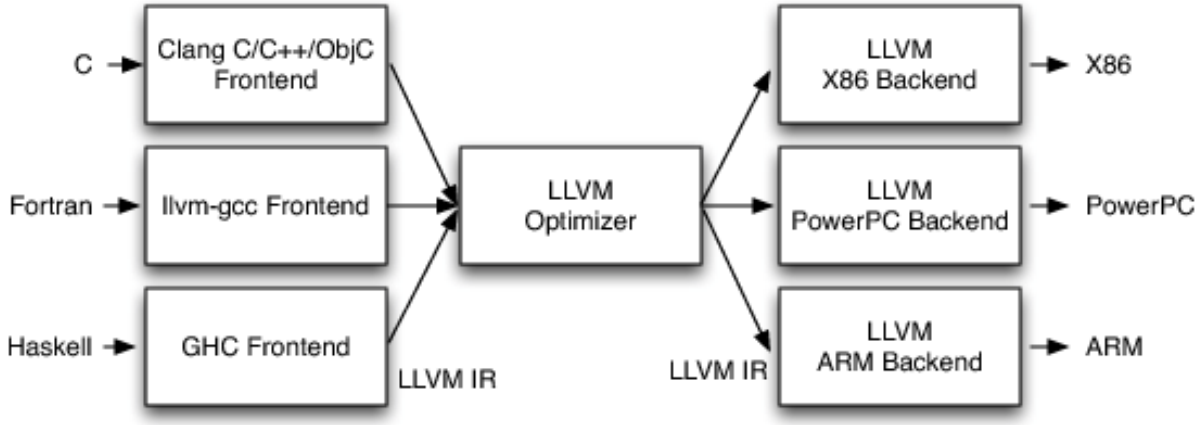

Figura 1.2 – Representação do uso de LLVM
Fonte: http://www.aosabook.org/en/llvm.html

Referências Bibliográficas

[1] The LLVM Compiler Infrastructure . https://llvm.org/. Accessed: 2017-09-26.

[2] WebAssembly. http://webassembly.org/. Accessed: 2017-10-16.

[3] Morgan Wilde. A brief introduction to LLVM. https://www.youtube.com/watch?v=a5-WaD8VV38, Jan 2016.