Construção de um compilador de Portugol para CLI usando Objective Caml

Matheus Meira Roberto
meira.matheus.rob@gmail.com

Faculdade de Computação
Universidade Federal de Uberlândia

17 de abril de 2017

Sumário

Capítulo 1

Introdução

Este relatório esta sendo construido com o objetivo de relatar os novos conhecimentos adquiridos na disciplina de compiladores.

1.1 Objetivo

O objetivo final é construir um compilador capaz de receber um código **Portugol** e gerar um **Assembly** que será executado pelo **CLR**(commom language runtime).

1.2 CLI

CLI (Common Intermediate Language) é o código assembly gerado para o CLR executar. Dessa forma, é necessârio instalar as ferramentas necessárias para criar os códigos em CLI e executá-los.

1.3 CLR e mono

No nosso trabalho, iremos utilizar o sistema operacional linux, sendo o **Ubuntu 14.04** a distribuição escolhida. Então, devemos instalar o **mono**. Pois, ele será o responsável por disponibilizar o CLR. O CLR é responsável por gerenciar a executar os programas.

1.3.1 Instalando o mono

Para instalar o mono é simples. Basta seguir os comando:

1. sudo apt-key adv –keyserver hkp://keyserver.ubuntu.com:80 –recv-keys 3FA7E0328081BFF6A14DA2

2. echo "deb http://download.mono-project.com/repo/debian wheezy main sudo tee /etc/apt/source xamarin.list
3. sudo apt-get update
4. sudo apt-get install mono-complete

1.4 Conjunto de instruções CLI

Dado que nosso objetivo é criar um compilador, é preciso entender melhor as instruçõs do CLI.

1. **add**, Add two values, returning a new value.
2. **add.ovf**, Add signed integer values with overflow check.
3. **add.ovf.un**, Add unsigned integer values with overflow check.
4. **and**, Bitwise AND of two integral values, returns an integral value.
5. **arglist**, Return argument list handle for the current method.
6. **beq <int32 (target)>**, Branch to target if equal.
7. **beq.s <int8 (target)>**, Branch to target if equal, short form.
8. **bge <int32 (target)>**, Branch to target if greater than or equal to.
9. **bge.s <int8 (target)>**, Branch to target if greater than or equal to, short form.
10. **bge.un <int32 (target)>**, Branch to target if greater than or equal to (unsigned or unordered).
11. **bge.un.s <int8 (target)>**, Branch to target if greater than or equal to (unsigned or unordered), short form.
12. **bgt <int32 (target)>**, Branch to target if greater than.
13. **bgt.s <int8 (target)>**, Branch to target if greater than, short form.
14. **bgt.un <int32 (target)>**, Branch to target if greater than (unsigned or unordered).
15. **bgt.un.s <int8 (target)>**, Branch to target if greater than (unsigned or unordered), short form.
16. **ble <int32 (target)>**, Branch to target if less than or equal to.
17. **ble.s <int8 (target)>**, Branch to target if less than or equal to, short form.
18. **ble.un <int32 (target)>**, Branch to target if less than or equal to (unsigned or unordered).

19. **ble.un.s <int8 (target)>**, Branch to target if less than or equal to (unsigned or unordered), short form.
20. **blt <int32 (target)>**, Branch to target if less than.
21. **blt.s <int8 (target)>**, Branch to target if less than, short form.
22. **blt.un <int32 (target)>**, Branch to target if less than (unsigned or unordered).
23. **blt.un.s <int8 (target)>**, Branch to target if less than (unsigned or unordered), short form.
24. **bne.un <int32 (target)>**, Branch to target if unequal or unordered.
25. **bne.un.s <int8 (target)>**, Branch to target if unequal or unordered, short form.
26. **box <typeTok>**, Convert a boxable value to its boxed form.
27. **br <int32 (target)>**, Branch to target.
28. **br.s <int8 (target)>**, Branch to target, short form.
29. **break**, Inform a debugger that a breakpoint has been reached.
30. **brfalse <int32 (target)>**, Branch to target if value is zero (false).
31. **brfalse.s <int8 (target)>**, Branch to target if value is zero (false), short form.
32. **brinst <int32 (target)>**, Branch to target if value is a non-null object reference (alias for brtrue).
33. **brinst.s <int8 (target)>**, Branch to target if value is a non-null object reference, short form (alias for brtrue.s).
34. **brnull <int32 (target)>**, Branch to target if value is null (alias for brfalse).
35. **brnull.s <int8 (target)>**, Branch to target if value is null (alias for brfalse.s), short form.
36. **brtrue <int32 (target)>**, Branch to target if value is non-zero (true).
37. **brtrue.s <int8 (target)>**, Branch to target if value is non-zero (true), short form.
38. **brzero <int32 (target)>**, Branch to target if value is zero (alias for brfalse).
39. **brzero.s <int8 (target)>**, Branch to target if value is zero (alias for brfalse.s), short form.
40. **call <method>**, Call method described by method.
41. **calli <callsitedescr>**, Call method indicated on the stack with arguments described by callsitedescr.
42. **callvirt <method>**, Call a method associated with an object.
43. **castclass <class>**, Cast obj to class.

44. **ceq**, Push 1 (of type int32) if value1 equals value2, else push 0.
45. **cgt**, Push 1 (of type int32) if value1 > value2, else push 0.
46. **cgt.un**, Push 1 (of type int32) if value1 > value2, unsigned or unordered, else push 0.
47. **ckfinite**, Throw ArithmeticException if value is not a finite number.
48. **clt**, Push 1 (of type int32) if value1 < value2, else push 0.
49. **clt.un**, Push 1 (of type int32) if value1 < value2, unsigned or unordered, else push 0.
50. **constrained. <thisType>**, Call a virtual method on a type constrained to be type T.
51. **conv.i**, Convert to native int, pushing native int on stack.
52. **conv.i1**, Convert to int8, pushing int32 on stack.
53. **conv.i2**, Convert to int16, pushing int32 on stack.
54. **conv.i4**, Convert to int32, pushing int32 on stack.
55. **conv.i8,**, Convert to int64, pushing int64 on stack.
56. **conv.ovf.i**, Convert to a native int (on the stack as native int) and throw an exception on overflow.

1.5 Primeiro programa em CIL

Para começarmos o projeto vamos compilar um código em cSharp e desassemblar para verificar o CLI gerado.

```
using System;

class HelloWorld {
  static void Main() {
    Console.WriteLine("Hello World");
  }
}
```

Inicialmente, utilizamos um comando para gerar um executável:

```
> mcs nomeDoArquivo.cs
```

Agora, para desassemblar o executável vamos utilizar um outro comando:

```
> monodis nomeDoArquivo.exe
```

Dessa forma, conseguimos obter o seguinte código CLI:

```
.class private auto ansi beforefieldinit HelloWorld
   extends [mscorlib]System.Object
  {

    // method line 1
    .method public hidebysig specialname rtspecialname
           instance default void '.ctor' ()  cil managed
    {
        // Method begins at RVA 0x2050
// Code size 7 (0x7)
.maxstack 8
IL_0000:  ldarg.0
IL_0001:  call instance void object::'.ctor'()
IL_0006:  ret
    } // end of method HelloWorld::.ctor

    // method line 2
    .method private static hidebysig
           default void Main ()  cil managed
    {
        // Method begins at RVA 0x2058
.entrypoint
// Code size 11 (0xb)
.maxstack 8
IL_0000:  ldstr "Hello World"
IL_0005:  call void class [mscorlib]System.Console::WriteLine(string)
IL_000a:  ret
    } // end of method HelloWorld::Main

  } // end of class HelloWorld
```

Como acabamos de gerar por comando o CLI, ele gera esse código que possui uma refência as linhas de onde cada comando esta no arquivo cSharp. De maneira mais organizada teríamos o seguinte código:

```
1 .assembly Hello {}
2 .assembly extern mscorlib {}
3 .method static void Main()
4 {
5
6   .entrypoint
7
8   .maxstack 1
9
10  ldstr "Hello, world!"
11
12  call void [mscorlib]System.Console::WriteLine(string)
13
14  ret
15
16 }
```

Na primeira linha temos o nome da classe:

```
> .assembly Hello {}
```

Na segunda ocorre a importação da biblioteca mscorlib, que será a responsável pela impressão na tela:

```
> .assembly extern mscorlib {}
```

E na terceira, temos a definição do método main

```
> .method static void Main()
```

Dentro do método temos o **.entrypoint** que é uma definição necessária e obrigatória. Depois, encontramos o **.maxstack 1** que é opcional e referencia o número de itens que deve ser seguido por uma ferramenta de análise. Em seguida encontramos o seguite comando:

```
> ldstr "Hello, world!"
```

Ela é responsável por adicionar a string na pilha. Já o penultimos comando imprime o que esta no topo da pilha. E a ultima linha encontramos **ret** responsável por delimitar o fim do código.