Contents

Acronyms

CIL Common Intermediate Language

CLI Common Language Infrastructure

CLS Common Language Specification

CTS Common Type System

IL Intermediate Language

ISO International Organization for Standardization

MSIL Microsoft Intermediate Language

VES Virtual Execution System

List of Figures

List of code listings

1 Introduction

1.1 The Common Language Infrastructure

The Common Language Infrastructure (CLI) is a specification developed by *Microsoft* and standardized by *ISO* and *ECMA International* [1, 2]. It describes executable code and an environment that allows numerous programming languages to be executed on various platforms.

The following four main aspects are covered by *the Common Language Infrastructure* [2]:

- ***The Common Type System (CTS)*** - a type system supporting types and operations found in many programming languages;
- **Metadata** - used for describing and referencing types defined by *the CTS*;
- ***The Common Language Specification (CLS)*** - an agreement between language designers and framework designers specifying a subset of *the CTS* and a set of usage conventions;
- ***The Virtual Execution System (VES)*** - responsible for loading and executing programs written for *the CLI*.

There are multiple implementations of *the CLI*, for instance: *.NET Framework*, *Shared Source Common Language Infrastructure Implementation*, *.NET Core* and *Mono*.

1.2 The Common Intermediate Language

The Common Intermediate Language (CIL) is also known as *the Microsoft Intermediate Language (MSIL)* or simply *the Intermediate Language (IL)* [3]. It is an object-oriented programming language that is a part of *the CLI*. Each language compatible with *the CLI* is compiled into *the CIL*. Moreover, *the CIL* is the actual language executed by *the VES*. The language is a reasonable compromise between user-friendly, high-level programming languages and low-level assemblers. Although *the CIL* is human-readable, it is still a stack-based language and writing programs manually is therefore quite difficult.

Technically, *the CIL* is just a set of over 200 instructions and it does not define a syntax for describing *the CLI metadata*. There is another syntax called *ILAsm* - an assembly language for *the CIL* [2]. However, many sources refer to *the CIL* and *ILAsm* interchangeably. Thus, *the CIL* means hereinafter the whole *ILAsm* syntax and *the CIL instruction set* refers to the correct meaning of *the Common Intermediate Language*.

```
1 .assembly extern mscorlib {}
2
3 .assembly HelloWorld {}
4
```

```
5  .method static public void main() cil managed
6  {
7    .entrypoint
8    .maxstack 1
9    ldstr "Hello world!"
10   call void [mscorlib]System.Console::WriteLine(string)
11   ret
12 }
```

Code listing 1.1: *Hello world* program

The difference between *ILAsm* and *the CIL* can be easily understood by analysing the *Hello world* program shown in code listing 1.1. Lines 1-8 and 12 of the example contain *the CLI* metadata described in *ILasm* syntax whereas lines 9-11 contain actual *CIL* instructions. The detailed meaning of each line can be explained as follows:

- Line 1 informs that an external assembly (`mscorlib`) should be loaded.
- Line 3 defines the assembly that should be created as the result of the compiler.
- Line 5 declares a static public method (`main`). It also defines that the method contains *CIL* code.
- Line 6 begins the body of the `main` method.
- Line 7 informs that the method should be used as the entry point of the program - it should be called by the compiler as the very first method.
- Line 8 specifies the maximal size of the evaluation stack associated with the method.
- Line 9 contains a `ldstr` instruction. It tells the compiler to put the *Hello world!* string onto the evaluation stack.
- Line 10 contains a `call` instruction. It tells the compiler to call the `WriteLine` method from the `System.Console` class defined in the `mscorlib` assembly.
- Line 11 contains a `ret` instruction. It tells the compiler to return the result of the method and to transfer the control to the next method on the call stack.
- Line 12 ends the body of the `main` method.

1.3 TUTAJ ZALOZENIA, CEL PRACY I ZAKONCZENIE WSTEPU

2 The CIL instruction set

In order to understand how the CIL instructions work, this section explains the concept of the global state and provides code examples introducing several CIL instructions. As the whole set contains more than 200 instructions, further information on each of them can be found in [2] and [1].

2.1 The global state

The CLI can manage multiple threads of control at the same time. A single thread of control can be thought as a call stack consisting of multiple method states. Although all the threads of control are rather independent, they still can access multiple managed heaps allocated in the shared memory space [2]. This concept is presented in figure 1.

A single method state includes several items required for *the VES* to execute the method [2], inter alia:

- an instruction pointer - it determines the next instruction that should be executed within the method;
- an evaluation stack - it stores evaluation values;
- a method information handle - it contains read-only method information;
- a local variable array;
- an argument array.

Generally, a single instruction can be understood as a description of how to manipulate the corresponding method state - there are only a few exceptions to this rule. Some instructions only require access to the evaluation stack while some do need to change other parts of the global state. To better understand these differences, the examples below present a couple of instructions and describe what effect they have on the global state.

2.2 The evaluation stack

As described above, a method state contains an evaluation stack. While a rich set of data types can be represented in memory, only a very limited subset of those types are supported by the CLI on an evaluation stack. This subset consists of the following data types [2]:

- `int32`,
- `int64`,
- `native int`,
- `F` (a floating-point number),

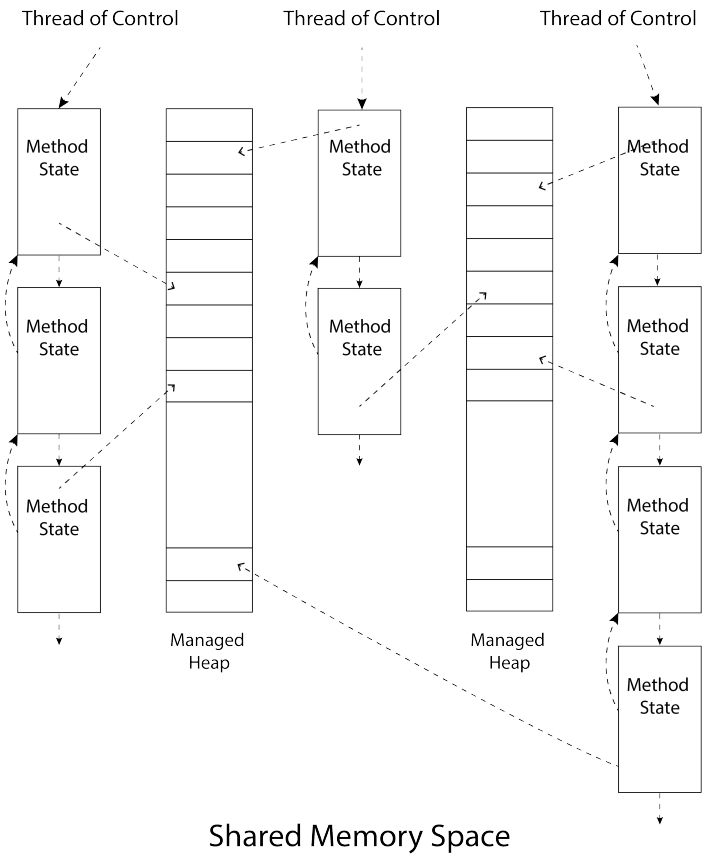

Figure 1: The global state concept

- `O` (an object reference),
- `&` (a managed pointer),
- `native unsigned int` (also an unmanaged pointer),
- a user-defined value type.

2.3 Stack transformations

A stack transformation can be thought as a sequence of pop and / or push operations performed on an evaluation stack. The majority of the CIL instructions uses such a sequence to get some values from the corresponding stack or to push some values onto it. In order to describe and visualise the effect of an instruction, the following syntax is used:

- S denotes an evaluation stack;
- $S \rightarrow S, v_1, v_2, ..., v_n$ denotes a stack transformation which is equivalent to pushing the values $v_1, v_2, ..., v_n$ onto the stack S;
- $S, u_1, u_2, ..., u_n \rightarrow S$ denotes a stack transformation which is equivalent to popping the values $u_1, u_2, ..., u_n$ from the stack S;
- $S, u_1, u_2, ..., u_n \rightarrow S, v_1, v_2, ..., v_m$ denotes a stack transformation which is equivalent to popping values $u_1, u_2, ..., u_n$ from the stack S and then pushing the values $v_1, v_2, ..., v_m$ onto it.

2.4 The examples

2.4.1 `ldc.i4.0`

The `ldc.i4.0` instruction pushes 0 onto the evaluation stack as `int32`. The corresponding stack transformation can be presented as follows [2]:

$$S \rightarrow S, 0$$

The program shown in code listing 2.1 uses the instruction to push 0 onto the stack and then writes out the top stack value (0) using the `call` instruction.

```
1  .assembly extern mscorlib {}
2
3  .assembly HelloWorld {}
4
5  .method static public void main() cil managed
6  {
7    .entrypoint
8    .maxstack 1
9    ldc.i4.0
```

```
10    call void [mscorlib]System.Console::WriteLine(int32)
11    ret
12  }
```

Code listing 2.1: Usage of `ldc.i4.0` instruction

The CIL provides a number of similar instructions to push other constant values onto the stack: `ldc.i4.1`, `ldc.i4.2`, `ldc.i4.3`, `ldc.i4.4`, `ldc.i4.5`, `ldc.i4.6`, `ldc.i4.7`, `ldc.i4.8` and `ldc.i4.m1` (or `ldc.i4.M1`) which pushes -1 onto the stack.

2.4.2 TUTAJ NASTEPNE PRZYKLADY UZYCIA INSTRUKCJI

3 TUTAJ ROZDZIAL O SEMANTYCE

4 The interpreter

The main part of this thesis is the interpreter which reads a CIL source code and is able to execute it.

TUTAJ KILKA ZDAN / AKAPITOW O TEORII INTERPRETACJI NA PODSTAWIE ZRODEL

4.1 The lexer and the parser

TUTAJ KILKA ZDAN O GRAMATYCE, TOKENIZACJI, I PARSOWANIU KODU DO AST

4.2 Technical background

In order to focus on the semantics of the CIL and not on data types, the interpreter has been written in `C#` which is itself compiled into the CIL so it uses the same type system and the same external assemblies.

The interpreter is a console application based on the `.NET Framework 4.7.2` so it can be executed in `Windows` operating system with the appropriate framework installed. The program can be run with the following command line argument:

- `--fileName` *`fileName`* - a path to a text file containing the source code to be interpreted.

The standard input, standard output and standard error streams are redirected to the console in order to allow the interpreted program to interact with the user. For instance, the program shown in code listing 1.1 calls the `Console.WriteLine` method to write out the *Hello world!* string to the standard output. The interpreter redirects the output therefore the string appears in the console. However, when an interpreter exception is caught, it is also written to the standard error stream.

4.3 The actual interpreter

Before interpreting the provided source code, the application checks if the code is supported by the interpreter. Once a non-supported feature is detected, one of the following exceptions is thrown:

- `InstructionNotSupportedException`,
- `FeatureNotSupportedException`.

Bibliography

[1] Information technology – Common Language Infrastructure (CLI). Standard ISO/IEC 23271:2012, The International Organization for Standardization, 2012. [Online] Available: `https://www.iso.org/standard/58046.html`.

[2] Common Language Infrastructure (CLI). Standard ECMA-335, ECMA International, 2012. [Online] Available: `https://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[3] A. Troelsen and P. Japikse. *Pro C# 7: With .NET and .NET Core*. Apress, 8th edition, 2017.