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.

```
.assembly extern mscorlib {}

.assembly HelloWorld {}

```

```
 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 ### TO DO: 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),

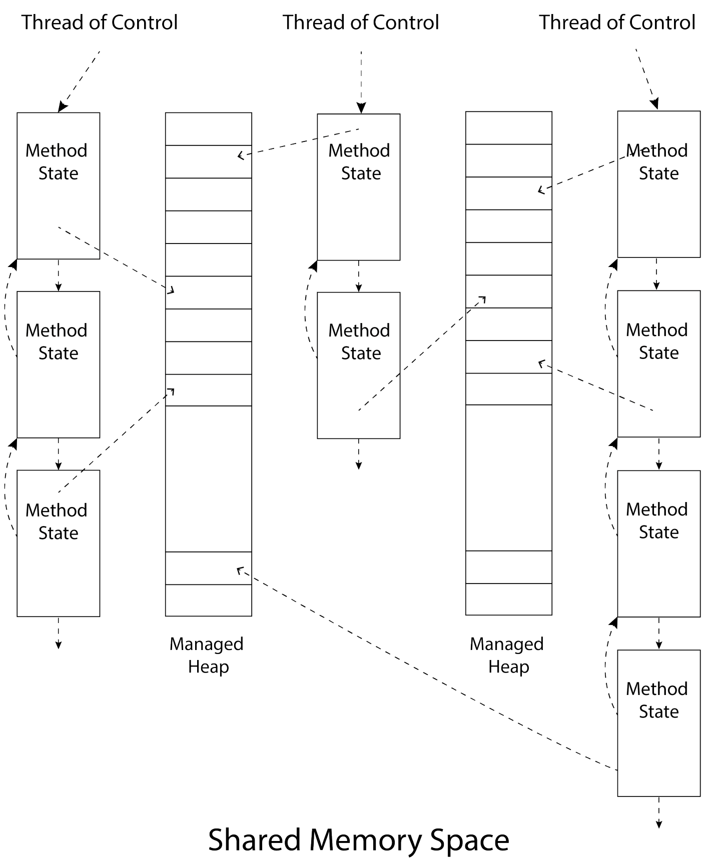

Figure 1: The global state concept.

- `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;
- $S \to S \cdot v_1 \cdot v_2 \cdot ... \cdot v_n$ denotes a stack transformation which is equivalent to pushing the values $v_1, v_2, ..., v_n$ onto the stack S;
- $S \cdot u_1 \cdot u_2 \cdot ... \cdot u_n \to S$ denotes a stack transformation which is equivalent to popping the values $u_1, u_2, ..., u_n$ from the stack S;
- $S \cdot u_1 \cdot u_2 \cdot ... \cdot u_n \to S \cdot v_1 \cdot v_2 \cdot ... \cdot v_m$ 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 Stack values

To illustrate the difference between various stack values, the following syntax represents the value v of the stack type `type`:

$$v_{\texttt{type}}$$

. For instance:

- $0_{\texttt{int32}}$ represents 0 of type `int32`.

2.5 Examples

2.5.1 `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]:

$$S \to S \cdot 0_{\texttt{int32}}$$

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.

```
.assembly extern mscorlib {}

.assembly HelloWorld {}

.method static public void main() cil managed
{
  .entrypoint
  .maxstack 1
  ldc.i4.0
  call void [mscorlib]System.Console::WriteLine(int32)
  ret
}
```

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.

2.5.2 ### TO DO: PRZYKLADY UZYCIA INNYCH INSTRUKCJI

3 The semantics

As described in 2.1, an instruction can be thought as a description of how to manipulate the current global state. Examples of such informal descriptions can be found in 2.5. [2] presents further examples but it does not provide any formal way to describe the CIL instructions and their semantics. Thus, one of the goals of the thesis is to introduce a formal, mathematical model to represent the semantics of the CIL.

The semantics presented below is patterned upon the example of the CIL semantics presented in [4]. However, this thesis attempts to simplify it and to represent the semantics on a higher level of abstraction.

3.1 The global state

The global state described in 2.1 can be represented as an ordered pair

$$\sigma = \langle \tau, \eta \rangle \tag{3.1}$$

where σ is a global state, τ is a set of states of the corresponding threads of control and η is a set of states of the managed heaps.

The following formula could be potentially used to describe the semantics [5]:

$$\langle I, \sigma_1 \rangle \to \sigma_2. \tag{3.2}$$

It can be understood as follows: if the program is in the state σ_1 and it executes the instruction I, the result state is σ_2. However, formula 3.2 treats the state and the current instruction as separate beings while the global state of the CIL contains the instruction pointer hence the formula can be simplified:

$$\sigma_1 \to \sigma_2. \tag{3.3}$$

.

3.2 The execution state

Since the global state and its mathematical representation can be very complex, this section is to provide a simpler concept called hereinafter *the execution state*. As a single instruction can only change a state of a single thread of control, there is no need to include the whole global state in the semantics. The execution state is then a part of the global state that could be potentially changed by an instruction and can be represented as an ordered pair:

$$\langle \gamma, \eta \rangle \tag{3.4}$$

where γ is the corresponding thread of control (its state) and η is a set of states of the managed heaps.

The semantics can be now presented as formulas of the following shape:

$$\langle \gamma_1, \eta_1 \rangle \to \langle \gamma_2, \eta_2 \rangle. \tag{3.5}$$

The formula has an analogous meaning to formula 3.3 however it operates on the execution state while the rest of the global state remains unchanged. For transparency, the angle brackets can be now omitted.

3.3 The method state

As mentioned before, the state of a single thread of control can be understood as a call stack of method states. Thus a similar syntax to the one introduced in section 2.3 is used to illustrate the changes made on a call stack. A single method state can be then described using a tuple

$$\langle I, S \rangle \tag{3.6}$$

where:

- I is the instruction pointer,
- S is the evaluation stack.

TO DO: ROZSZERZYĆ OPIS METHOD STATE'U O POZOSTAŁE ELEMENTY

3.4 The final syntax

Taken all together, the semantics of the CIL can be represented as a set of formulas of the following shape:

$$\gamma_1 \cdot \mu_{1,1} \cdot \ldots \cdot \mu_{1,n}, \eta_1 \rightarrow \gamma_2 \cdot \mu_{2,1} \cdot \ldots \cdot \mu_{2,m}, \eta_2 \tag{3.7}$$

where

- γ_1, γ_2 are call stacks,
- $\mu_{1,1}, \ldots, \mu_{1,n}, \mu_{2,1}, \ldots, \mu_{2,m}$ are method states,
- η_1, η_2 are sets of managed heaps.

For instance, the following describes a transformation which pushes 0 of type `int32` onto the evaluation stack of the top method state:

$$\gamma \cdot \langle I, S \rangle, \eta \rightarrow \gamma \cdot \langle I, S \cdot 0_{\texttt{int32}} \rangle, \eta.$$

3.5 Instruction pointers

To show what actually instruction is pointed by the instruction pointer I, the following formula is used:

$$I \nearrow \texttt{instruction}. \tag{3.8}$$

No special mathematical syntax is used to describe various instruction types. For example, the following formula means that the instruction pointer I points to a `ldc.i4.0` instruction:

$$I \nearrow \texttt{ldc.i4.0}.$$

Additionally, according to the syntax presented in [4], I_n denotes hereinafter a pointer to the n-th instruction of a method.

3.6 Examples

This section follows the informal descriptions contained in 2.5 and uses the conventions described above in order to specify the semantics of several CIL instructions which can be found in formulas 3.9-3.10.

$$\frac{I_n \nearrow \texttt{ldc.i4.0}}{\gamma \cdot \langle I_n, S\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot 0_{\texttt{int32}}\rangle, \eta} \tag{3.9}$$

$$\frac{I_n \nearrow \texttt{ldc.i4.1}}{\gamma \cdot \langle I_n, S\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot 1_{\texttt{int32}}\rangle, \eta} \tag{3.10}$$

TO DO: UZUPELNIC SEMANTYKA KOLEJNYCH INSTRUKCJI ZGODNIE Z SEKCJA 2.5

4 The interpreter

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

TO DO: KILKA ZDAN / AKAPITOW O TEORII INTERPRETACJI NA PODSTAWIE ZRODEL

4.1 The lexer and the parser

TO DO: 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 Detection of non-supported features

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 and written out to the console:

- `InstructionNotSupportedException`,
- `FeatureNotSupportedException`.

4.4 The actual interpreter

In order to implement the interpreter, a couple of concepts described before needed to be modelled in `C#`. For example, the `CilExecutionState` class is responsible for the execution state explained in 3.4 and can be seen in listing 4.1. There are other classes which have been implemented by analogy, inter alia:

- `CilControlState`,
- `CilMethodState`,
- `CilEvaluationStack`.

```
public class CilExecutionState
{
  public CilControlState ControlState { get; }

  public CilManagedMemory ManagedMemory { get; }

  public CilExecutionState(CilControlState controlState,
     CilManagedMemory managedMemory)
  {
    ControlState = controlState;
    ManagedMemory = managedMemory;
  }
}
```

Code listing 4.1: The `CilExecutionState` class.

4.4.1 The visitor

Since the semantics presented in 3 bases on transformations of the execution state, the same concept has been used to implement the interpreter. A single transformation of the state always depends on the current instruction pointer so the process of interpreting can be thought as visiting various instructions presented in the abstract syntax tree. To implement such a behaviour, a design pattern called *Visitor* has been used. The pattern allows to write a new operation for each element of an object structure [6].

`CilInstructionsVisitor` shown in listing 4.2 is an abstract class which implements the Visitor design pattern. In order to make its implementation reusable, the class does not decide what instruction should be visited next. Instead, an abstract method `GetNextInstruction` is used. As the CIL instruction set contains around 200 instructions, the visitor itself has been split up into several smaller classes (called subvisitors) responsible for visiting instructions of different types. Each instruction has its corresponding abstract method in the subvisitors set that is called by the main `Visit` method once an instruction of the appropriate type is going to be visited.

Using such an abstract implementation allows to implement different types of visitors for the abstract syntax tree. For now, there is only one implementation - the `CilInterpreterInstructionsVisitor` class and its subvisitors. This implementation uses the execution state to implement the `GetNextInstruction` method. To be precise, the instruction pointer of the top method state is used to determine the next instruction to visit.

```
public abstract class CilInstructionsVisitor
{
  protected abstract InstructionNoneVisitor InstructionNoneVisitor { get; }

  protected abstract InstructionMethodVisitor InstructionMethodVisitor { get; }

  protected abstract InstructionStringVisitor InstructionStringVisitor { get; }

  protected abstract InstructionIVisitor InstructionIVisitor { get; }

  protected abstract InstructionTypeVisitor InstructionTypeVisitor { get; }

  protected abstract InstructionVarVisitor InstructionVarVisitor { get; }

  protected abstract InstructionRVisitor InstructionRVisitor { get; }

  protected abstract InstructionBrVisitor InstructionBrVisitor { get; }

  protected abstract InstructionFieldVisitor InstructionFieldVisitor { get; }

  protected abstract InstructionI8Visitor InstructionI8Visitor { get; }

  protected abstract InstructionSwitchVisitor InstructionSwitchVisitor { get; }

  protected abstract InstructionTokVisitor InstructionTokVisitor { get; }

  public void Visit()
  {
    var nextInstruction = GetNextInstruction();
    while (nextInstruction != null)
    {
      VisitInstruction(nextInstruction);
      nextInstruction = GetNextInstruction();
    }
  }

  protected abstract CilInstruction GetNextInstruction();

  private void VisitInstruction(CilInstruction instruction)
  {
    if (instruction is CilInstructionNone instructionNone)
      InstructionNoneVisitor.VisitInstructionNone(instructionNone);
    else if (instruction is CilInstructionMethod instructionMethod)
      InstructionMethodVisitor.VisitInstructionMethod(instructionMethod);
    else if (instruction is CilInstructionString instructionString)
      InstructionStringVisitor.VisitInstructionString(instructionString);
    else if (instruction is CilInstructionI instructionI)
      InstructionIVisitor.VisitInstructionI(instructionI);
    else if (instruction is CilInstructionType instructionType)
      InstructionTypeVisitor.VisitInstructionType(instructionType);
    else if (instruction is CilInstructionVar instructionVar)
      InstructionVarVisitor.VisitInstructionVar(instructionVar);
    else if (instruction is CilInstructionR instructionR)
      InstructionRVisitor.VisitInstructionR(instructionR);
    else if (instruction is CilInstructionBr instructionBr)
      InstructionBrVisitor.VisitInstructionBr(instructionBr);
    else if (instruction is CilInstructionField instructionField)
      InstructionFieldVisitor.VisitInstructionField(instructionField);
    else if (instruction is CilInstructionI8 instructionI8)
      InstructionI8Visitor.VisitInstructionI8(instructionI8);
    else if (instruction is CilInstructionSwitch instructionSwitch)
      InstructionSwitchVisitor.VisitInstructionSwitch(instructionSwitch);
    else if (instruction is CilInstructionTok instructionTok)
      InstructionTokVisitor.VisitInstructionTok(instructionTok);
    else
      throw new ArgumentException($"CIL instruction cannot be recognized: '{instruction.ToString()}'.");
  }
}
```

Code listing 4.2: The `CilInstructionsVisitor` class.

4.5 Examples

Having the abstract visitors structure described in the previous section, implementing the interpreter is equivalent to completing the implementation of all of the abstract methods of the subvisitors. The implementation itself is based on the semantics described in 3.

4.5.1 `ldc.i4.0`

The `VisitLoadConstI40Intruction` method shown in listing 4.3 corresponds to the `ldc.i4.0` instruction described in 2.5.1 with its semantics presented in formula 3.9. The meaning of the method can be explained as follows:

- Line 3 creates a new stack value of type `int32` with value equal to `0`.
- Line 4 pushes the newly created value to the top of the evaluation stack. `ControlState.EvaluationStack` is just a shorthand for accessing the evaluation stack of the top method state.
- Line 6 moves the instruction pointer of the top method state to the next instruction of the same method. The `MoveToNextInstruction` method is just a helper as such an operation is used very often.

```
1  protected override void VisitLoadConstI40Intruction(
       LoadConstI40Instruction instruction)
2  {
3    var stackVal = new CilStackValueInt32(0);
4    ControlState.EvaluationStack.Push(stackVal);
5
6    ControlState.MoveToNextInstruction();
7  }
```

Code listing 4.3: The `VisitLoadConstI40Intruction` method.

TO DO: INNE PRZYKLADY IMPLEMENTACJI METOD TYPU VISIT

5 Testing

TO DO: OPIS TESTOWANIA, W SZCZEGOLNOSCI OPIS APLIKACJI / SRODOWISKA STWORZONEGO NA TE POTRZEBY

6 Limitations

TO DO: OPIS OGRANICZEN, KTORE TYCZA SIE SAMEGO INTERPRETERA JAK I SRODOWISKA

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.

[4] W. Zychla. *exTensible Multi Security: Security Framework for .NET*. PhD thesis, Institute of Computer Science, University of Wrocław, 2008.

[5] T. Wierzbicki. *Języki programowania*. 2001.

[6] Johnson R. Gamma E., Helm R. and Vlissides J. *Design Patterns: Elements of reusable Object-Oriented Software*. Pearson Education, 1994.