Contents

Acronyms

AST Abstract syntax tree

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 Tables

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 ### 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]. A full list of the instructions is also shown in tables 1 and 2 which illustrate whether each instruction was implemented in the final interpreter or not.

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`,

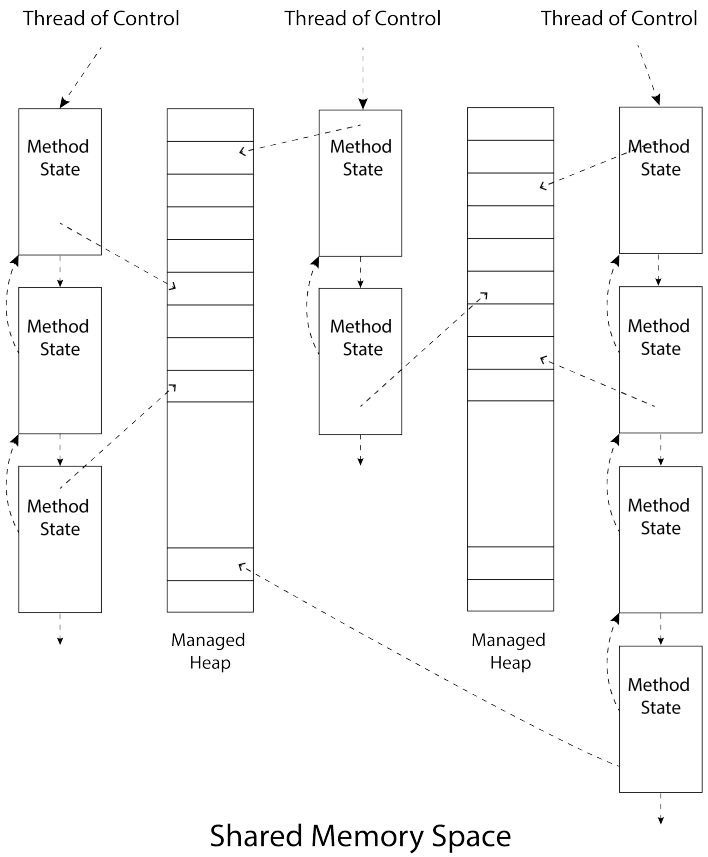

Figure 1: The global state concept.

- `F` (a floating-point number),
- `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;
- $S \rightarrow S \cdot v_1 \cdot v_2 \cdot ... \cdot v_n$ denotes a stack transformation which is equivalent to pushing the values $v_1, v_2, ..., v_n$ onto the stack S;
- $S \cdot u_1 \cdot u_2 \cdot ... \cdot u_n \rightarrow S$ denotes a stack transformation which is equivalent to popping the values $u_1, u_2, ..., u_n$ from the stack S;
- $S \cdot u_1 \cdot u_2 \cdot ... \cdot u_n \rightarrow S \cdot v_1 \cdot v_2 \cdot ... \cdot v_m$ 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]:

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

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.

```
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 the `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 `stloc.0` and `ldloc.0`

The `stloc.0` instruction pops a value v from the evaluation stack and moves it to the 0-th position in the local variable array. Its stack transformation can be represented as follows [2]:

$$S \cdot v \rightarrow S$$

Conversely, the `ldloc.0` instruction loads the 0-th local variable v onto the evaluation stack and its stack transformation is [2]:

$$S \rightarrow S \cdot v$$

The program shown in code listing 2.2 uses both instruction to store 7 in the 0-th local variable and then to load it onto the stack.

There are several similar instructions provided by the CIL: `stloc.1`, `stloc.2`, `stloc.3`, `stloc`, `stloc.s`, `ldloc.1`, `ldloc.2`, `ldloc.3`, `ldloc`, `ldloc.s`.

```
1  .assembly extern mscorlib {}
2
3  .assembly HelloWorld {}
4
5  .method static public void main() cil managed
6  {
7    .entrypoint
8    .maxstack 1
9    .locals init ([0] int32 n)
10   ldc.i4.7
```

```
11    stloc.0
12    ldloc.0
13    call void [mscorlib]System.Console::WriteLine(int32)
14    ret
15  }
```

Code listing 2.2: Usage of the `stloc.0` and `ldloc.0` instructions.

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. Furthermore, the formal CIL semantics cannot be easily found among other sources. 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 Code verifiability

A full CLI implementation should involve a verification algorithm that is responsible for checking the correctness of instructions and local variables. Precisely, the algorithm requires all local variables to be initialised before executing a method. It also simulates all possible control flows and branches in order to verify each reachable instruction. Obviously, the algorithm cannot predict the actual values on the current evaluation stack but is still able to check their number and types [5].

Since the algorithm itself is responsible for checking types, the semantics assumes all the values on the evaluation stack and local variables to be correct and verified. The interpreter described later was also implemented to handle valid programs only.

3.2 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 [6]:

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

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:

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

.

3.3 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:

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

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.

3.4 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

$$\langle I, S, M, L, A \rangle \tag{3.6}$$

where:

- I is the instruction pointer,
- S is the evaluation stack,
- M is the method information handle,
- L is the local variable array,
- A is the argument array.

3.5 The final syntax

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

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

where

- γ_1, γ_2 are call stacks,
- $\mu_{1,1}, ..., \mu_{1,n}, \mu_{2,1}, ..., \mu_{2,m}$ 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:

$$\gamma \cdot \langle I, S, M, L, A \rangle, \eta \to \gamma \cdot \langle I, S \cdot 0_{\texttt{int32}}, M, L, A \rangle, \eta.$$

3.6 Instruction pointers

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

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

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:

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

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

3.7 Arrays

A single method state contains 2 different arrays: a local variable array and an argument array. An array A can be represented as a function associating a single value to each natural number that is less than the length of the array. In order to formalise the changes made on arrays, the following term is used:

$$A[n \mapsto v] \tag{3.9}$$

It can be understood as the array A with the n-th element set to the value v. Formally [4]:

$$(A[n \mapsto v])(i) = \begin{cases} v & \text{if } i = n, \\ A(i) & \text{otherwise.} \end{cases} \tag{3.10}$$

3.8 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 3.11-??.

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.0}}{\gamma \cdot \langle I_n, S, M, L, A \rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 0_{\texttt{int32}}, M, L, A \rangle, \eta} \tag{3.11}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.1}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 1_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.12}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.2}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 2_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.13}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.3}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 3_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.14}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.4}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 4_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.15}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.5}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 5_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.16}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.6}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 6_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.17}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.7}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 7_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.18}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.8}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot 8_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.19}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.m1}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot (-1)_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.20}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.M1}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot (-1)_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.21}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4.s}\ v}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot v_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.22}$$

$$\frac{I_n \rightsquigarrow \texttt{ldc.i4}\ v}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S \cdot v_{\texttt{int32}}, M, L, A\rangle, \eta} \tag{3.23}$$

$$\frac{I_n \rightsquigarrow \texttt{stloc.0}}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \to \gamma \cdot \langle I_{n+1}, S, M, L[0 \mapsto v], A\rangle, \eta} \tag{3.24}$$

$$\frac{I_n \rightsquigarrow \texttt{stloc.1}}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L[1 \mapsto v], A\rangle, \eta} \tag{3.25}$$

$$\frac{I_n \rightsquigarrow \texttt{stloc.2}}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L[2 \mapsto v], A\rangle, \eta} \tag{3.26}$$

$$\frac{I_n \rightsquigarrow \texttt{stloc.3}}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L[3 \mapsto v], A\rangle, \eta} \tag{3.27}$$

$$\frac{I_n \rightsquigarrow \texttt{stloc.s}\ i}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L[i \mapsto v], A\rangle, \eta} \tag{3.28}$$

$$\frac{I_n \rightsquigarrow \texttt{stloc}\ i}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L[i \mapsto v], A\rangle, \eta} \tag{3.29}$$

$$\frac{I_n \rightsquigarrow \texttt{ldloc.0}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot L(0), M, L, A\rangle, \eta} \tag{3.30}$$

$$\frac{I_n \rightsquigarrow \texttt{ldloc.1}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot L(1), M, L, A\rangle, \eta} \tag{3.31}$$

$$\frac{I_n \rightsquigarrow \texttt{ldloc.2}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot L(2), M, L, A\rangle, \eta} \tag{3.32}$$

$$\frac{I_n \rightsquigarrow \texttt{ldloc.3}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot L(3), M, L, A\rangle, \eta} \tag{3.33}$$

$$\frac{I_n \rightsquigarrow \texttt{ldloc.s}\ i}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot L(i), M, L, A\rangle, \eta} \tag{3.34}$$

$$\frac{I_n \rightsquigarrow \texttt{ldloc}\ i}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot L(i), M, L, A\rangle, \eta} \tag{3.35}$$

$$\frac{I_n \rightsquigarrow \texttt{nop}}{\gamma \cdot \langle I_n, S, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L, A\rangle, \eta} \tag{3.36}$$

$$\frac{I_n \rightsquigarrow \texttt{dup}}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S \cdot v \cdot v, M, L, A\rangle, \eta} \tag{3.37}$$

$$\frac{I_n \rightsquigarrow \texttt{pop}}{\gamma \cdot \langle I_n, S \cdot v, M, L, A\rangle, \eta \rightarrow \gamma \cdot \langle I_{n+1}, S, M, L, A\rangle, \eta} \tag{3.38}$$

4 The interpreter

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

The process of interpreting source code is commonly split up into several phases [7]:

- *Lexical analysis* is the initial step of processing the input source code. It reads the text and divides it into specific tokens.
- *Syntax analysis* or *parsing* - this step produces an AST based on the list of tokens.
- *Interpreting* - the last phase processes the AST in order to evaluate expressions and execute statements.

Since the thesis focuses mainly on the last step, the interpreter bases on an external library called `Irony` which allows to tokenise and parse the input source code in a very straightforward way. The library requires a definition of a context-free grammar that should be used to process the input code. An almost complete grammar of the CIL is provided in [2] thus it was used as a sample. For instance, code listing 4.1 presents a rule implemented within the interpreter that corresponds to the one defined in the original grammar:

```
name1:  id | DOTTEDNAME | name1 '.'  name1;
```

```
1  name1.Rule =
2    id |
3    DOTTEDNAME |
4    name1 + _(".") + name1;
```

Code listing 4.1: An example of the grammar definition within the interpreter.

The whole grammar implementation can be found in the repository, under the path:

`/CILantro/Parsing/CilGrammar.cs`.

4.1 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 source code of the interpreter can be found under the following link:

https://github.com/kvasnyk/CILantro/tree/master/CILantro.

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.

4.2 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`.

4.3 The actual interpreter

In order to implement the interpreter, a couple of concepts described before needed to be modelled in `C#`. While the semantics is an abstract description of the language that defines each instruction as a transformation of the execution state, it does not provide any details on how to implement the transformations or the execution state itself. This very chapter describes various approaches that could be followed to reflect some elements of the semantics in the implementation of an interpreter.

4.3.1 The evaluation stack

Potentially, the evaluation stack could be implemented as a simple stack that provides methods to pop and push values. There is the `Stack` class in the `C#` standard library that meets these requirements. However, there are a couple of problems that need to be resolved while using such a model:

- many CIL instructions require more than one value to be popped or pushed at the same time;
- the evaluation stack can only operate on a very limited set of types;

- the CIL requires specific conversions between the standard types and the stack types while pushing or popping values.

In order to avoid such problems, the evaluation stack has been implemented as a new class containing a private `Stack` instance. The class provides a number of useful *pop* and *push* methods that can operate on the limited set of stack types as well as other standard types. While calling a method, all the required conversions are made if possible or an exception is thrown. Additionally, the `CilEvaluationStack` class allows to call a method that pops or pushes several values at the same type. The source code of the class can be seen in the repository, under the following path:

`/CILantro/Interpreting/State/CilEvaluationStack.cs`.

4.3.2 The managed memory

The semantics operates on a set of managed heaps. This thesis does not go into the detail and collectively treats all the managed heaps as *the managed memory*. Ideally, the managed memory could be implemented by directly using the unmanaged memory. Such a solution is undoubtedly the most flexible and the most desired while implementing any interpreter.

Nevertheless, operating on the unmanaged memory might be really challenging using `C#` as it is based on the managed memory itself. Alternatively, a simpler approach can be adopted - the managed memory can be simply simulated as a dictionary mapping addresses to objects. Using such an approach has its serious consequences - no unmanaged memory operations are allowed and the managed memory depends on the `C#` managed memory.

As this thesis does not focus on the managed memory but on the semantics of the CIL, the managed memory has been implemented using the simpler method described above. The source code of the `CilDictionaryManagedMemory` can be found in the repository, under the path:

`/CILantro/Interpreting/Memory/CilDictionaryManagedMemory.cs`

However, in order to make the implementation easy to change, there is an abstract class `CilManagedMemory` that could be potentially used to implement the managed memory in any other way, including the desired solution that involves the unmanaged memory.

4.3.3 The state arrays

There are 2 similar arrays included in a single method state: the local variable array and the argument array. They cannot be implemented as standard arrays as their elements must be accessible by a local variable name or an argument name respectively. In order to fulfil this criterion, the interpreter uses the `CilOrderedDictionary` class. It bases on the `OrderedDitionary` class from the `C#` standard library that allows to access its elements by an index as well as a name. The class can be found in the repository, under the following path:

`/CILantro/Interpreting/State/CilOrderedDictionary.cs`

4.3.4 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 [8].

`CilInstructionsVisitor` is an abstract class which implements the Visitor design pattern. Its source code can be found in the repository, under the following path:

`/CILantro/Visitors/CilInstructionsVisitor.cs`.

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.

4.4 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.

4.4.1 `ldc.i4.0`

The `VisitLoadConstI40Intruction` method shown in listing 4.2 corresponds to the `ldc.i4.0` instruction described in 2.5.1 with its semantics presented in formula 3.11. 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 4.2: The `VisitLoadConstI40Intruction` method.

4.4.2 `stloc.0`

The `VisitStoreLocal0Instruction` method is shown in listing 4.3. It is related to the `stloc.0` instruction described in 2.5.2 and its semantics presented in formula 3.24. The implementation can be understood as follows:

```
1  protected override void VisitStoreLocal0Instruction(
       StoreLocal0Instruction instruction)
2  {
3    var localType = ControlState.Locals.GetLocalType(null, 0);
4    ControlState.EvaluationStack.PopValue(_program, localType,
         out var value);
5    ControlState.Locals.Store(null, 0, value);
6
7    ControlState.MoveToNextInstruction();
8  }
```

Code listing 4.3: The `VisitSetLocal0Instruction` method.

5 Testing

Writing an interpreter of such a complex programming language as the CIL may be a little complicated. Implementing all the visit instructions at the same time seems to be impossible so a specific methodology needed to be arranged in order to achieve the desired effect. As a common approach in software development, an iterative methodology was introduced. Since the main goal of the thesis was to create an interpreter and maximise its compatibility with the CIL specification, the methodology was based on the following assumption - the main goal of each iteration is to increase the compatibility of the interpreter and the coverage of the CIL instruction set. The methodology can be also described as a specific process consisting of 4 steps:

1. Write a test.
2. Cover the implementation of the interpreter so that the test passes.
3. Run all the existing tests.
4. Correct the implementation so that all the tests pass.
5. Repeat steps 1-4.

The process is illustrated in figure 2.

5.1 The tests

Potentially, a set of unit tests could be written based on the documentation and the semantics of each instruction. Nevertheless, such a task might be very difficult so another type of tests was used instead. Since there are functioning implementations of the CLI, working of the interpreter can be compared to working of one of such implementations. To be exact, the `.NET Framework 4.7.2` was selected to write the tests.

Taking into account the abilities of the interpreter, only console programs writing to the standard input, reading from the standard outuput and using the standard error stream are considered proper tests. The vast majority of the tests are written in `C#`. Using the `ildasm` tool provided together with the `.NET Framework`, the tests can be decompiled to their CIL source code which can be then used as the input file for the interpreter.

Once a test is prepared, it can be executed using the interpreter and the result of the execution can be compared to the result of the original test program. The result of such a comparison is assumed to be correct when the output produced by the interpreter is the same as the output produced by the original program for the same input. Even though a single test can be potentially executed using an infinite set of different inputs, only a small part is used so that the process is not too time-consuming. In order to make the process as reliable as possible, a set of random inputs is prepared every time the test is executed.

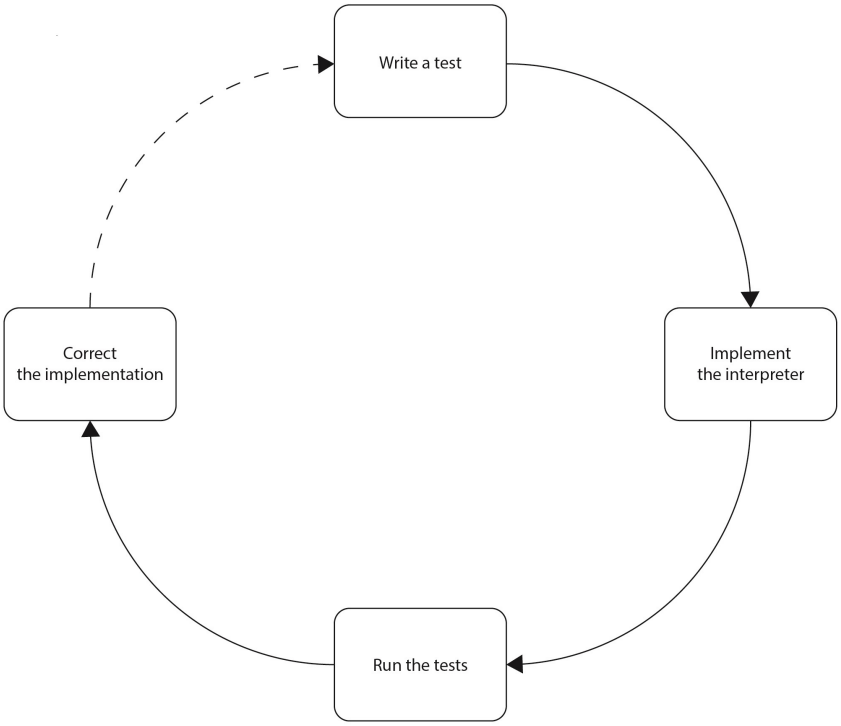

Figure 2: The process of implementing the interpreter.

5.2 The environment

Since repeating the process described above manually might be arduous for bigger numbers of tests, a dedicated system was written separately to automate the procedure. The system is hereinafter called *the tools* or *the environment*. It comprises 2 projects:

- a back-end application,
- a front-end application.

The back-end application is built using `ASP.NET Core` which is a modern framework for building Web APIs and applications [9]. It provides a set of API actions and socket connections that were designed to handle tests, their categories and executions. The application uses a database to store all the data collected so far.

The front-end application was written in `Typescript` using `React.js` which is a popular library created for building user interfaces [10]. It is a single-page application that interacts with users and communicates with the back-end application in order to send or receive data.

The tools provide several functionalities described below.

5.2.1 Finding tests

The environment watches the folder provided in the configuration and displays a list of the new tests that have been found. Each of the found tests can be then saved in the database and used to perform various operations. An example of the functionality is shown in figure 3.

5.2.2 Managing categories

The tools allow the user to define categories and subcategories that can be listed, ordered and filtered. The categories and the subcategories can be then assigned to the tests. The *Categories* page is shown in figure 4.

5.2.3 Managing tests

The tests can be also listed, ordered and filtered. Each test is presented as a single card of a specific colour which determines the state of the test. If a card is grey, the related test is not configured and needs some additional attention - the card contains a list of tasks to be done in order to configure the test properly. If a card is purple, the corresponding test has been marked as unsupported. If a card is white, the test is ready to be used and run. Additionally, if a test has been already run, the card shows an icon that represents the result of the last run of the test. The *Tests* page is shown in figure 5.

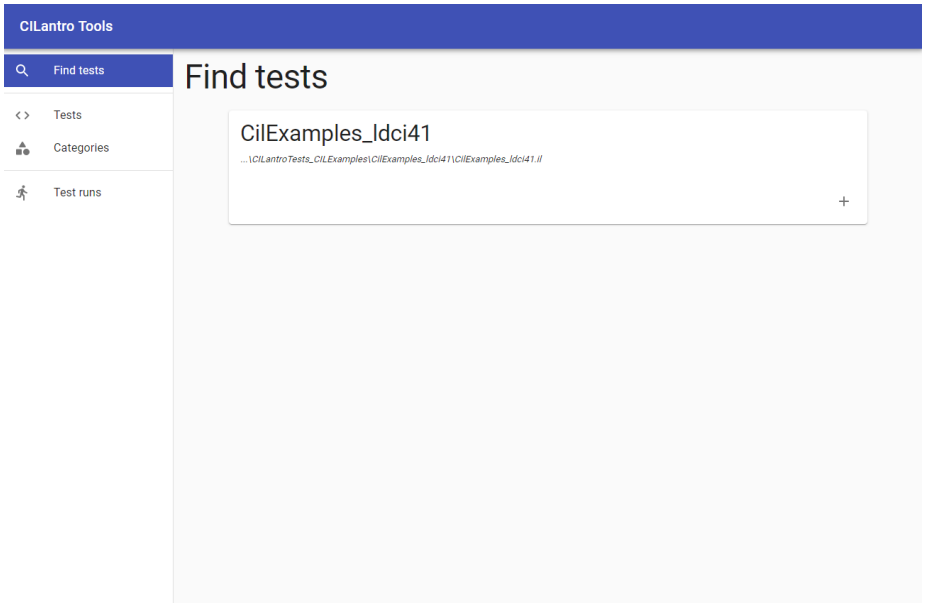

Figure 3: The environment - the *Find tests* functionality.

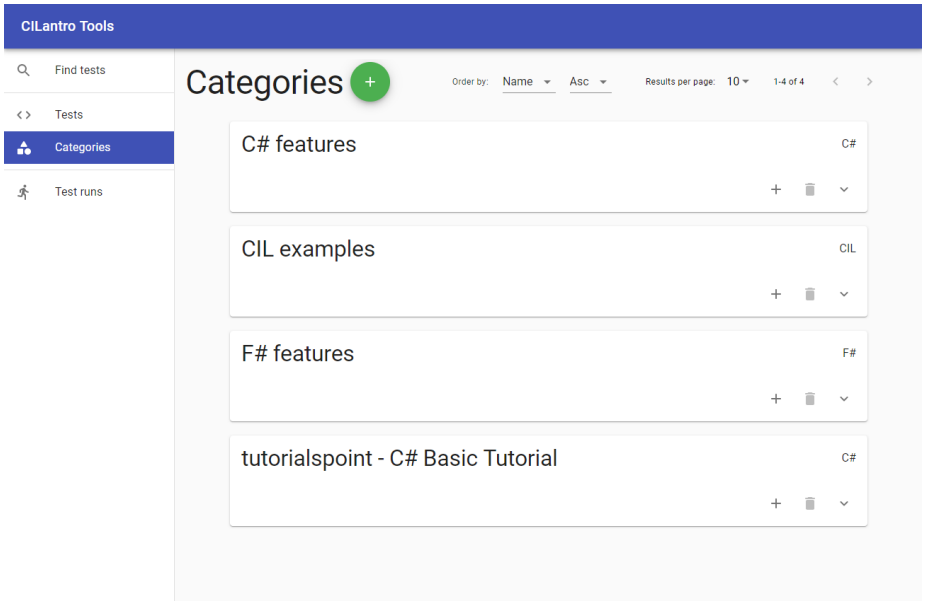

Figure 4: The environment - the *Categories* page.

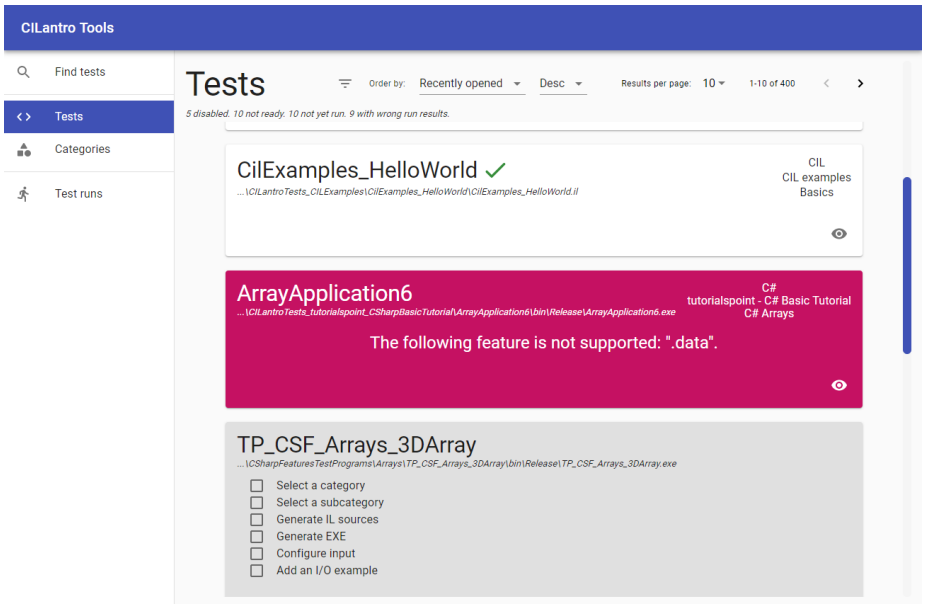

Figure 5: The environment - the *Tests* page.

5.2.4 Managing a single test

The details of each test defined in the application can be shown on a separate page. The page is divided into the following tabs:

- `OVERVIEW` - it shows some basic information about the test such as its category and subcategory;
- `IL SOURCES` - allows the user to display and regenerate the main `.il` file containing the CIL source code of the test;
- `EXE` - can be used to compile the source code into an executable file required to execute the test;
- `I/O` - used for defining example inputs and the pattern of input accepted by the test.

Above the tabs, there is an action area that allows the user to:

- run the original program, the interpreter or both at the same time;
- schedule a test run for the test;
- mark the test as supported or non-supported.

Figure 6 shows a single test page while figure 7 presents an example of executing both the original program and the interpreter using the comparison view.

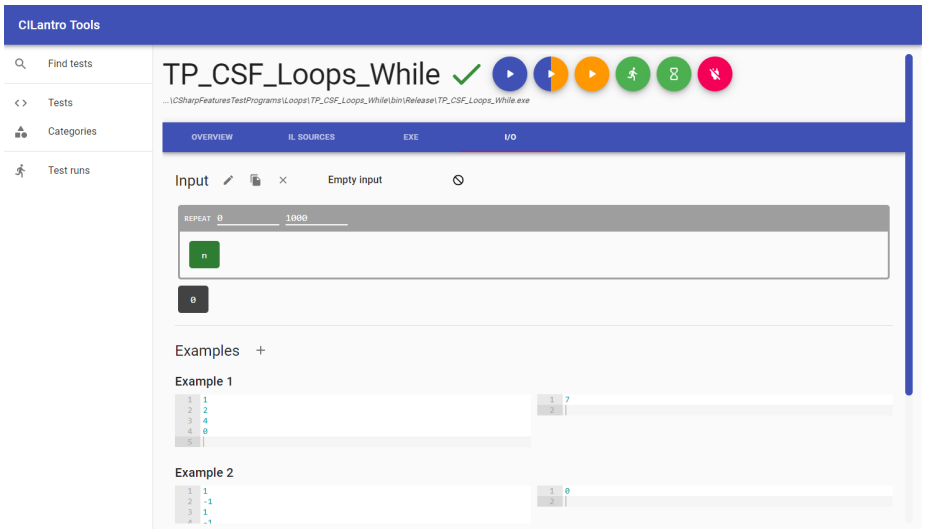

Figure 6: The environment - an example test page.

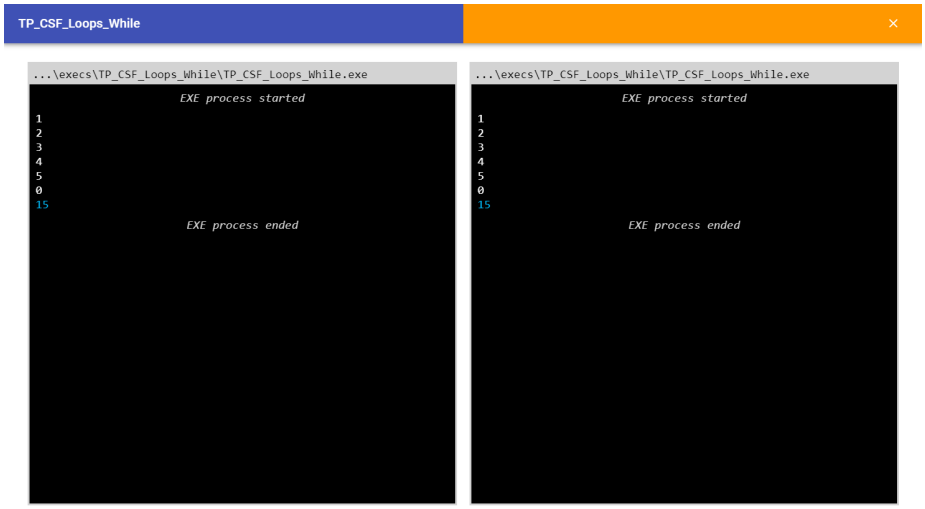

Figure 7: The environment - running a test in the comparison view.

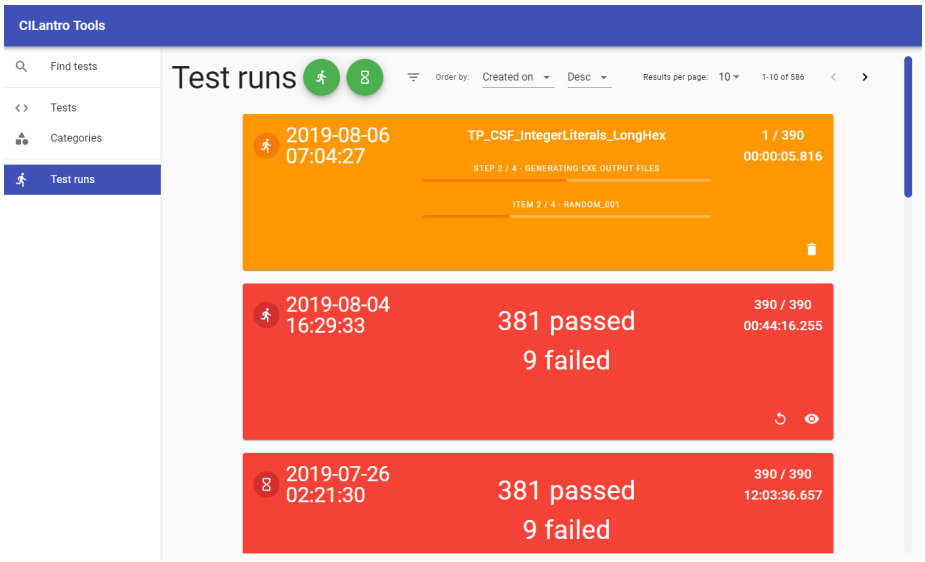

Figure 8: The environment - the *Test runs* page.

5.2.5 Managing test runs

The test runs can be listed, ordered and filter. A single run is represented by a card of a specific colour. If a card is orange, the corresponding run is being executed at the moment. If a card is red, the run has already ended but its result is wrong. If a card is green, the related run has ended and the result is correct. Each run can be also restarted and the user can display its details.

The page contains two buttons that start a single test run including all the well-configured tests. There are two types of runs: quick and full. A quick run executes each test using only 3 random inputs while a full run uses 100 random inputs for each test. Regardless of the type, each run also executes the tests using the inputs defined in the tools. The *Test runs* page is shown in figure 8.

5.2.6 Managing a single test run

Once a test run is finished, its details can be viewed on a separate page. The page contains a card for each test that has been executed within the test run. A card is red if the corresponding test ended with a wrong result and green otherwise. The environment allows the user to open a card and access the details of the executed test. An open card shows a table where each input is represented by a single row containing icons that show the result of each of the following execution phases:

- generating input,
- executing the original program,

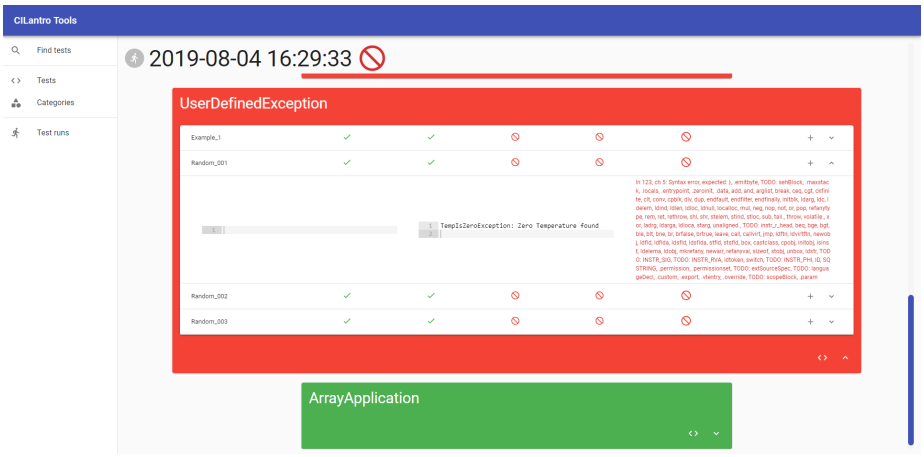

Figure 9: The environment - the *Test run* functionality.

- executing the interpreter,
- comparing the results.

Additionally, each row can be also opened in order to see the input and the outputs of the original program and the interpreter. An example of the functionality is shown in figure 9.

6 Results

nop	○	pop	○	ldind.u4	○
break	○	jmp	○	ldind.i8	○
ldarg.0	○	call	○	ldind.i	○
ldarg.1	○	calli	○	ldind.r4	○
ldarg.2	○	ret	○	ldind.r8	○
ldarg.3	○	br.s	○	ldind.ref	○
ldloc.0	○	brfalse.s	○	stind.ref	○
ldloc.1	○	brtrue.s	○	stind.i1	○
ldloc.2	○	beq.s	○	stind.i2	○
ldloc.3	○	bge.s	○	stind.i4	○
stloc.0	○	bgt.s	○	stind.i8	○
stloc.1	○	ble.s	○	stind.r4	○
stloc.2	○	blt.s	○	stind.r8	○
stloc.3	○	bne.un.s	○	add	○
ldarg.s	○	bge.un.s	○	sub	○
ldarga.s	○	bgt.un.s	○	mul	○
starg.s	○	ble.un.s	○	div	○
ldloc.s	○	blt.un.s	○	div.un	○
ldloca.s	○	br	○	rem	○
stloc.s	○	brfalse	○	rem.un	○
ldnull	○	brtrue	○	and	○
ldc.i4.m1	○	beq	○	or	○
ldc.i4.0	○	bge	○	xor	○
ldc.i4.1	○	bgt	○	shl	○
ldc.i4.2	○	ble	○	shr	○
ldc.i4.3	○	blt	○	shr.un	○
ldc.i4.4	○	bne.un	○	neg	○
ldc.i4.5	○	bge.un	○	not	○
ldc.i4.6	○	bgt.un	○	conv.i1	○
ldc.i4.7	○	ble.un	○	conv.i2	○
ldc.i4.8	○	blt.un	○	conv.i4	○
ldc.i4.s	○	switch	○	conv.i8	○
ldc.i4	○	ldind.i1	○	conv.r4	○
ldc.i8	○	ldind.u1	○	conv.r8	○
ldc.r4	○	ldind.i2	○	conv.u4	○
ldc.r8	○	ldind.u2	○	conv.u8	○
dup	○	ldind.i4	○	callvirt	○

Table 1: The CIL instructions covered by the interpreter (part 1 of 2).

cpobj	○	ldelem.i	○	sub.ovf.un	○
ldobj	○	ldelem.r4	○	endfinally	○
ldstr	○	ldelem.r8	○	leave	○
newobj	○	ldelem.ref	○	leave.s	○
castclass	○	stelem.i	○	stind.i	○
isinst	○	stelem.i1	○	conv.u	○
conv.r.un	○	stelem.i2	○	arglist	○
unbox	○	stelem.i4	○	ceq	○
throw	○	stelem.i8	○	cgt	○
ldfld	○	stelem.r4	○	cgt.un	○
ldflda	○	stelem.r8	○	clt	○
stfld	○	stelem.ref	○	clt.un	○
ldsfld	○	ldelem	○	ldftn	○
ldsflda	○	stelem	○	ldvirtftn	○
stsfld	○	unbox.any	○	ldarg	○
stobj	○	conv.ovf.i1	○	ldarga	○
conv.ovf.i1.un	○	conv.ovf.u1	○	starg	○
conv.ovf.i2.un	○	conv.ovf.i2	○	ldloc	○
conv.ovf.i4.un	○	conv.ovf.u2	○	ldloca	○
conv.ovf.i8.un	○	conv.ovf.i4	○	stloc	○
conv.ovf.u1.un	○	conv.ovf.u4	○	localloc	○
conv.ovf.u2.un	○	conv.ovf.i8	○	endfilter	○
conv.ovf.u4.un	○	conv.ovf.u8	○	unaligned.	○
conv.ovf.u8.un	○	refanyval	○	volatile.	○
conv.ovf.i.un	○	ckfinite	○	tail.	○
conv.ovf.u.un	○	mkrefany	○	Initobj	○
box	○	ldtoken	○	constrained.	○
newarr	○	conv.u2	○	cpblk	○
ldlen	○	conv.u1	○	initblk	○
ldelema	○	conv.i	○	no.	○
ldelem.i1	○	conv.ovf.i	○	rethrow	○
ldelem.u1	○	conv.ovf.u	○	sizeof	○
ldelem.i2	○	add.ovf	○	Refanytype	○
ldelem.u2	○	add.ovf.un	○	readonly.	○
ldelem.i4	○	mul.ovf	○		
ldelem.u4	○	mul.ovf.un	○		
ldelem.i8	○	sub.ovf	○		

Table 2: The CIL instructions covered by the interpreter (part 2 of 2).

7 Conclusions and future work

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] Lidin S. *Expert .NET 2.0 IL Assembler*. Apress, 2006.

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

[7] Torben Ægidius Mogensen. *Basics of Compiler Design*. 2010.

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

[9] N. Barbettini. *the little ASP.NET Core book*. 2017.

[10] A. Banks and E. Porcello. *Learning React: Functional Web Development with React and Redux*. O'Reilly Media, 1st edition, 2017.