NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Conversion of CLI stack-based intermediate language to a register-based representation

Aristotelis G. Koutsouridis

Supervisor: **Yannis Smaragdakis,** Professor

ATHENS

MAY 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετατροπή της βασισμένης σε στοίβα CLI ενδιάμεσης γλώσσας, σε μια αναπαράσταση βασισμένη σε καταχωρητές

Αριστοτέλης Γ. Κουτσουρίδης

Επιβλέπων: **Γιάννης Σμαραγδάκης**, Καθηγητής

ΑΘΗΝΑ

ΜΑΪΟΣ 2020

BSc THESIS

Conversion of CLI stack-based intermediate language to a register-based representation

Aristotelis G. Koutsouridis

S.N.: 1115201600078

SUPERVISOR: **Yannis Smaragdakis,** Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Μετατροπή της βασισμένης σε στοίβα CLI ενδιάμεσης γλώσσας, σε μια αναπαράσταση βασισμένη σε καταχωρητές

Αριστοτέλης Γ. Κουτσουρίδης
Α.Μ.: 1115201600078

ΕΠΙΒΛΕΠΩΝ: **Γιάννης Σμαραγδάκης,** Καθηγητής

ABSTRACT

We briefly present the Virtual Execution System (VES) of the Common Language Infrastructure (CLI) and the properties of the stack based Common Intermediate Language (CIL). Then we inspect managed assemblies by browsing through types, type members and CIL method instructions. A Control Flow Graph (CFG) is generated for method instructions. This CFG method representation is used as a first approach to convert the stack-based intermediate representation to a register-based (linear) representation. A second approach to this conversion is examined which uses important properties of managed CIL code. Finally, we compare our results and highlight future applications of register-based IL in code analysis.

SUBJECT AREA: Compilers

KEYWORDS: .NET Framework, Intermediate Representation, Virtual Execution System, Code Analysis, Operand Stack

ΠΕΡΙΛΗΨΗ

Παρουσιάζουμε συντόμως το Εικονικό Περιβάλλον Εκτέλεσης (VES) του Common Language Infrastructure (CLI) standard, καθώς και τις ιδιότητες της βασισμένης σε στοίβα ενδιάμεσης γλώσσας Common Intermediate Language (CIL). Έπειτα, αναλύουμε τους τύπους και τις μεθόδους του CIL κώδικα. Αρχικά, κατασκευάζουμε μια αναπαράσταση του γράφου ροής ελέγχου για να μετατρέψουμε την βασισμένη σε στοίβα ενδιάμεση γλώσσα σε μια γραμμική αναπαράσταση με καταχωρητές. Επίσης, δοκιμάζουμε μια δεύτερη, πιο απλή, προσέγγιση όπου η μετατροπή γίνεται βάση των ιδιοτήτων και των περιορισμών του κώδικα. Τέλος, συγκρίνουμε τα αποτελέσματα μας, επισημαίνοντας τις εφαρμογές τους στην ανάλυση προγραμμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μεταγλωττιστές

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: .NET Framework, Στοίβες, Ενδιάμεση Γλώσσα, Ανάλυση Προγραμμάτων, Γράφος Ροής Ελέγχου

ACKNOWLEDGEMENTS

Για τη διεκπεραίωση της παρούσας Πτυχιακής Εργασίας, θα ήθελα να ευχαριστήσω τους επιβλέποντες, καθ. Γιάννη Σμαραγδάκη, Ηλία Τσατίρη και Σίφη Λαγουβάρδο για τη συνεργασία και την πολύτιμη βοήθεια στην ολοκλήρωση της.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

Η εργασία αυτή μελετά μερικά ζητήματα στην επιστημονική περιοχή των μεταγλωττιστών και της ανάλυσης προγραμμάτων. Παρόλα αυτά, έχει γίνει μια προσπάθεια ωστε ο αναγνώστης να μή χρειάζεται εκ των προτέρων γνώση σχετικά με το αντικείμενο. Για την απλή κατανόηση των αποτελεσμάτων της εργασίας, αρκούν βασικές γνώσεις αντικειμενοστραφή προγραμματισμού, κάποιας γλώσσας assembly καθώς και εμπειρία με διαδεδομένες τεχνικές προγραμματισμού και δομές δεδομένων.

1. INTRODUCTION

Common Intermediate Language is the product of compilation of code written in high-level CLI-compliant languages like C# or F#. Once compiled, code written in one of these languages produces a binary consisting of CIL code. This binary file format is known as the Portable Executable (PE) format. Unlike native languages like C which compile down to machine code, CIL code is managed by a Virtual Execution System (VES). The VES uses a Just-In-Time (JIT) compiler which converts CIL code to machine code which can be run on a CPU. CIL code may be low level compared to CLI-compliant languages but it is far from machine code. It is an object oriented, stack-based bytecode language.

1.1 Common Language Infrastructure

The Common Language Infrastructure (CLI) provides a **specification** for executable code and the execution environment (the Virtual Execution System) in which it runs. Executable code is presented to the VES as modules. A module is a single file containing executable content. The CLI specification covers four main areas:

- **The Common Type System (CTS)**: The CTS provides a rich type system that supports the types and operations found in many programming languages. The CTS is intended to support the complete implementation of a wide range of programming languages.
- **Metadata**: The CLI uses metadata to describe and reference the types defined by the CTS. Metadata is stored (that is, persisted) in a way that is independent of any particular programming language.
- **The Common Language Specification (CLS)**: The CLS is an agreement between language designers and framework (that is, class library) designers. It specifies a subset of the CTS and a set of usage conventions.
- **The Virtual Execution System (VES)**: The VES implements and enforces the CTS model. The VES is responsible for loading and running programs written for the CLI. It provides the services needed to execute managed code and data, using the metadata to connect separately generated modules together at runtime (late binding).

The goal of such an architecture is to describe executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures [1]. It is worth noting that CLI is a specification/standard, not an implementation. The following software contain implementations of the CLI:

- **.NET Framework** is Microsoft's original commercial implementation of the CLI.
- **.NET Core** is the free and open-source multi-platform successor to .NET Framework, released under the MIT license.
- **.NET Compact Framework** is Microsoft's commercial implementation of the CLI for portable devices and Xbox 360.

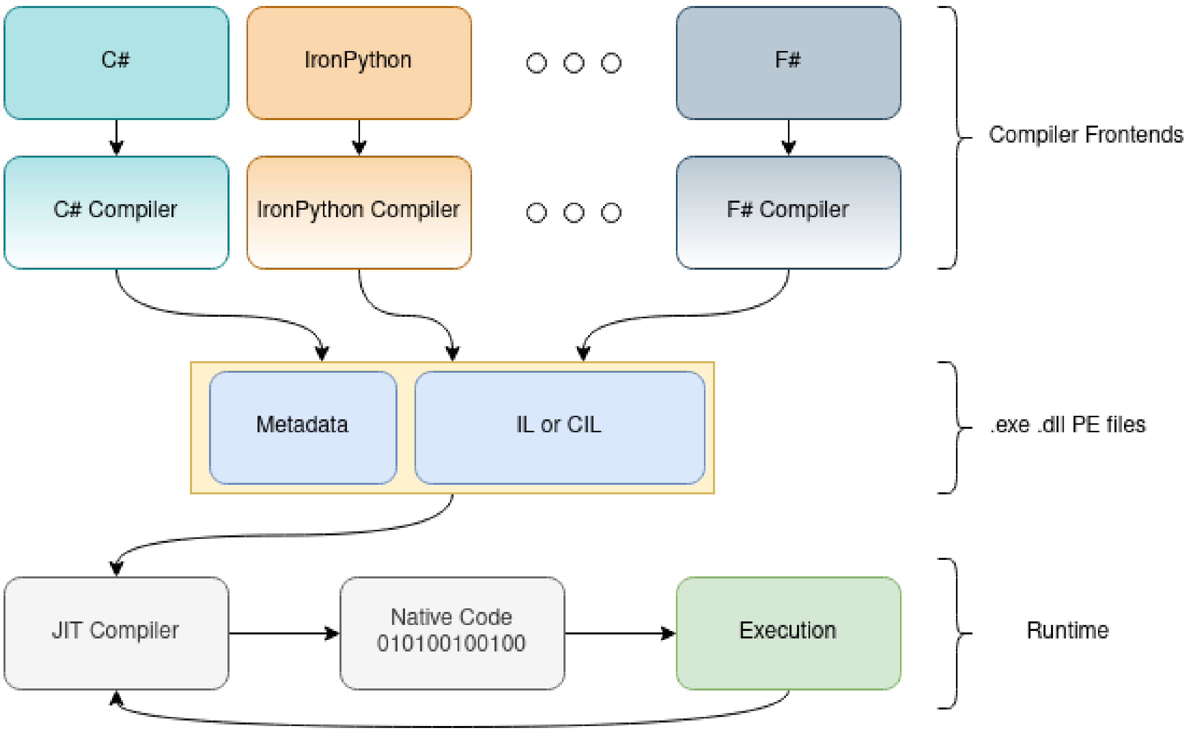


Figure 1.1: Architecture of CLI

- **.NET Micro Framework** is an open source implementation of the CLI for resource-constrained devices.
- **Mono** is an alternative open source implementation of the CLI. It is designed to allow developers to easily create cross platform applications.

Aside from implementing the runtime environment, these frameworks provide developers with useful tooling and class libraries. In later chapters we use such tools and libraries of the **Mono** project in order to inspect assemblies.

1.2 Virtual Execution System

The Virtual Execution System, also called Execution Engine, provides an environment for the execution of managed CIL code. This environment includes additional services including memory management, type safety, exception handling, garbage collection, security and thread management.

1.2.1 Data Types

When describing a language or a runtime, it is common practice to start from the building blocks: builtin data types. The following table describes the basic data types that are supported directly by the CLI. These data types can be manipulated with CIL code through the instruction set. We will discuss later the way code is executed in the VES.

Note that native-size types (*native int*, *native unsigned int*, *O* and *&*) which vary in size depending on the underlying architecture, don't have a specific size until JIT compilation/runtime. This happens because the architecture becomes known when the CLI is

Table 1.1: Data types supported by the CLI

int8	8-bit two's-complement signed value
unsigned int8	8-bit unsigned binary value
int16	16-bit two's-complement signed value
unsigned int16	16-bit unsigned binary value
int32	32-bit two's-complement signed value
unsigned int32	32-bit unsigned binary value
int64	64-bit two's-complement signed value
unsigned int64	64-bit unsigned binary value
float32	32-bit IEC 60559:1989 floating-point value
float64	64-bit IEC 60559:1989 floating-point value
native int	native size two's-complement signed value
native unsigned int	native size unsigned binary value, also unmanaged pointer
F	native size floating-point number (internal to VES, not user visible)
O	native size object reference to managed memory
&	native size managed pointer (can point into managed memory)

initialized. This implies that field and stack frame offsets are also not known at compile time.

The & data type (managed pointer) is similar to the O type, but points 'inside' of the object. That is, a managed pointer is allowed to point to a field within an object or an element within an array, rather than to point to the 'start' of object or array. In order to allow managed pointers to be used more flexibly, they are also permitted to point to areas that aren't under the control of the CLI garbage collector, such as static variables, and unmanaged memory.

1.2.2 Machine State

We will describe the components of the VES in terms of machine state. As mentioned earlier, the VES supports thread management. The CLI manages multiple threads of control, heaps and shared memory. A thread of control can be thought of as a list of method states. A method call links a new method state to the list and a returning method deletes a method state from the list. This resembles the common model of stack based calling sequence that we encounter in many environments. The method state can contain object references pointing to the heaps or shared memory. Figure 1.2 illustrates the VES managing the threads.

1.2.3 Method State

The CLI method state resembles the traditional **invocation stack frame** of languages like C/C++. The method state consists of the following components:

- **Instruction Pointer (IP)**: This points to the next CIL instruction to be executed by the CLI in the present method. This pointer is similar to the program counter of any assembly language.

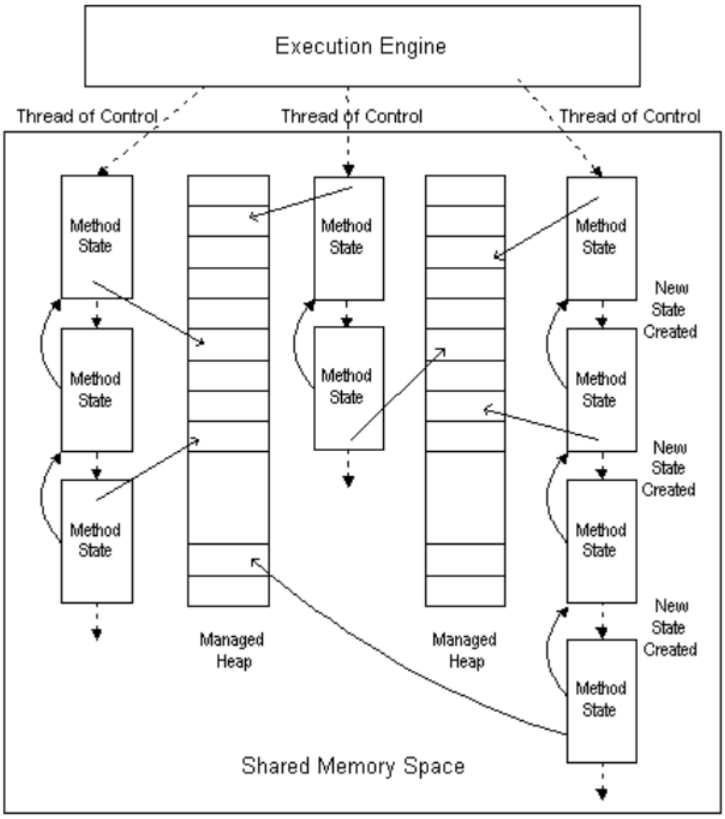


Figure 1.2: Machine State model

- **Evaluation Stack**: The stack is empty upon method entry. Its contents are entirely local to the method and are preserved across call instructions (that's to say, if this method calls another, once that other method returns, our evaluation stack contents are "still there"). At all times it is possible to deduce which one of a reduced set of types(see table 1.1) is stored in any stack location at a specific point in the CIL instruction stream.

- **Local Variable Array**: Values of local variables are preserved across calls (in the same sense as for the evaluation stack). A local variable can hold any data type.

- **Argument Array**: The values of the current method's incoming arguments (starting at index 0). These can be read and written by logical index.

- **MethodInfo Handle**: This contains read-only information about the method. In particular it holds the signature of the method, the types of its local variables, and data about its exception handlers.

- **Local Memory Pool**: The CLI includes instructions for dynamic allocation of objects from the local memory pool (localloc). The memory allocated in the local memory pool is reclaimed upon method context termination.

- **Return State Handle**: This handle is used to restore the method state on return from the current method.

- **Security Descriptor**: This descriptor is used by the CLI security system to record security overrides (assert, permit-only, and deny).

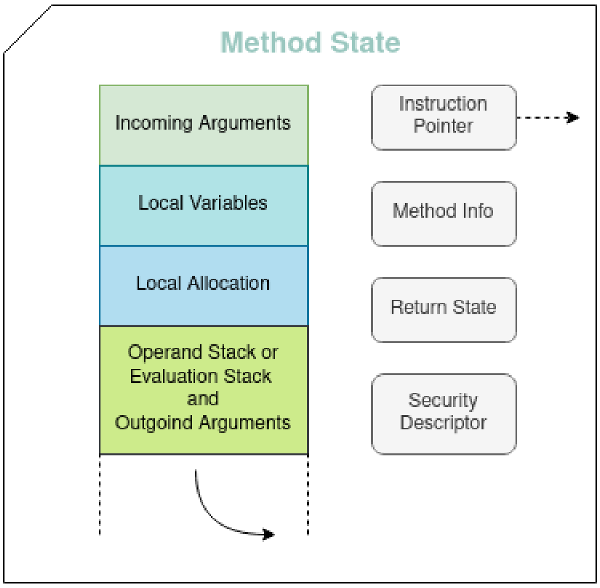


Figure 1.3: Method state model. The local allocation region is present because the CLI allows for stack allocation aside from heap allocation. The outgoing arguments are part of the evaluation stack because they are pushed/popped with respect to the instruction set. The illustrated stack grows on the bottom when elements are pushed to it.

```
# Suppose r10 and r11 contain the address of a and b in memory
lw $r1, 0($r10)     # Load a from memory into register r1
lw $r2, 0($r11)     # Load b from memory into register r1
add $r3, $r1, $r2   # Calculate a+b into r3
sub $r4, $r1, $r2   # Calculate a-b into r4
mul $r3, $r3, $r4   # Calculate (a+b)(a-b) into r3

```

Figure 1.4: Low level MIPS assembly code

We will concentrate on the execution environment of a method because methods contain the CIL code we are interested in and more specifically we will describe the evaluation stack in detail, as it is a crucial part of the conversion algorithms.

1.2.4 Evaluation Stack

Common CPU instruction set architectures describe a set of instructions performing reads or writes on a set of finite registers. Each instruction of a program is associated with some register addresses. Usually, two of the addresses are input registers(they provide the input data for the instruction) and a single register is the output address. In this case the instruction can be thought of as a binary operator. A simple and common example is the addition instruction. Two register addresses are specified as input to the instruction and the sum of the contents of those registers is written to the output register address. Consider the MIPS assembly code in figure 1.4, which loads a and b from some location in memory and calculates the expression $(a + b)(a - b)$:

We observe that this architecture relies on saving intermediate results in registers. A **stack machine**, in contrast, saves the intermediate results of instructions in a stack as the name hints. Let's examine figure 1.5 which contains the equivalent CIL code of the MIPS instructions in figure 1.4.

```
1  // Starting from an empty stack.
2  ldarg.1      // Load(push) 'a' from a memory location onto the stack
3  ldarg.2      // Load(push) 'b' from a memory location onto the stack
4  add          // Pop the two elements off the stack, push the sum on the stack.
5  ldarg.1      // Push 'a' again
6  ldarg.2      // Push 'b' again
7  sub          // Pop the two elements, push the difference (a-b) on the stack.
8  mul          // Multiply the two elements of the stack: (a+b) and (a-b)
9  // Now the stack contains only one element: the result.
10
```

Figure 1.5: CIL code that calculates the expression $(a+b)(a-b)$

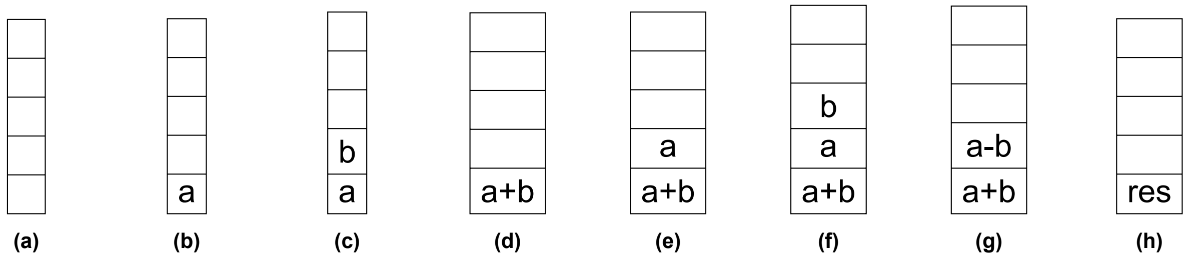


Figure 1.6: State of the evaluation stack before each instruction of figure 1.5

Initially the stack is empty. After each instruction, values may be popped/pushed from/to the evaluation stack. The first instruction loads the value a from an arbitrary address and pushes a on the stack. Now, the evaluation stack contains only one element: a. The next instruction does the same for b. So after the first two instructions the stack contains 2 elements: b at the top of the stack and a at the bottom(due to the LIFO behaviour of a stack). The *add* instruction at line 3 consumes these two elements of the stack by popping them which again results in an empty stack and then calculates the sum $a+b$. The result is pushed to the empty stack resulting in a stack with a single element: $a+b$. Next, 2 load instructions follow, which will have the same result as the first two loads, except now, the bottom of the stack has an element which holds the result $a+b$. In the same fashion as the *add* instruction the *sub* instruction at line 7 will pop the loaded values and push the difference on top of the result of the addition. Now the stack contains two elements. Finally, the last instruction will pop both elements off the stack and push the product $(a+b)(a-b)$. In this example the state of the stack before each instruction execution is shown in figure 1.6.

The elements of the stack can be of any basic type(*int64, int32, O, &* ...) as well as value types. Most CLI instructions retrieve their arguments from the evaluation stack and place their return values on the stack. **The type state of the stack (the stack depth and types of each element on the stack) at any given point in a program shall be identical for all possible control flow paths**. This is a very useful property that we will consider in later chapters. For example, a program that has a loop which pushes a value to the stack in each iteration is invalid CIL code.

The evaluation stack is used also to perform method calls. Before a method is called with a list of arguments, the arguments are loaded onto the stack. The *call* instruction then copies(or shares) the arguments to the incoming arguments array of the called method. Once the method returns, it places the return value on the evaluation stack of the calling method. This mechanism allows an arbitrary number of method arguments and a single or no return values.

1.3 CIL Instruction Set

Instructions perform operations on data. The data types that instructions work with, were presented in table 1.1. Each instruction is described by the following parts [4]:

- An **instruction variant table** describing the binary format, assembly language notation, and description of each variant of the instruction.
- A **stack transition diagram**, that describes the state of the evaluation stack before and after the instruction is executed.
- A description of the **instruction semantics**.
- A list of **exceptions** that might be thrown by the instruction(3 exceptions which can be thrown by any instruction are omitted: *System.ExecutionEngineException, System.StackOverflowException, System.OutOfMemoryException*)
- A section describing the **verifiability** conditions associated with the instruction.

We will omit details about the binary format of instructions. We will rely on libraries(*Mono.Cecil*) in order to inspect CIL code and therefore instructions. For our purposes, the instruction binary format can be abstracted. *Mono.Cecil* will provide us with a high-level representation of programs and instructions. It is worth noting though, that the binary format of the intermediate language of CLI has 1-byte opcodes and optional parameters. This design serves the **compactness** of code.

1.3.1 Instruction Variant Table

The instruction variant table describes the binary and assembly format of an instruction as well as a short description. Table 1.2 illustrates the instruction variant table for *ldarga* instruction which loads a method argument from the argument array onto the evaluation stack.

Table 1.2: Instruction variant table for the *ldarga* and *ldarga.s* instructions

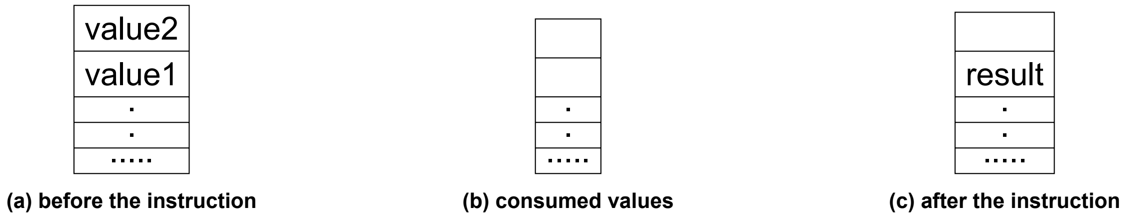
Format	**Assembly Format**	**Description**
FE 0A <unsigned int16>	ldarga *argNum*	Fetch the address of argument *argNum*.
0F <unsigned int8>	ldarga.s *argNum*	*ldarga* short form.

These tables give us good information about each instruction, but the don’t provide a formal description of the instruction’s behaviour. This is done by the stack transition diagram.

1.3.2 Stack Transition Diagram

The stack transition diagram displays the state of the evaluation stack before and after the instruction is executed. The following, is a typical stack transition diagram:

$$..., value1, value2 \quad \longrightarrow \quad ..., result$$



(a) before the instruction (b) consumed values (c) after the instruction

Figure 1.7: Evaluation stack states

This diagram indicates that the stack shall have at least two elements on it, and in the definition the topmost value (“top-of-stack” or “most-recently-pushed”) will be called value2 and the value underneath (pushed prior to value2) will be called value1. In diagrams like this, the stack grows to the right, across the page. The instruction removes these values from the stack and replaces them by another value, called result. Figure 1.7 illustrates this procedure in three steps.

Table A.3 of the appendix, lists the entire instruction set, together with control flow and stack behaviour values for each instruction (control flow is examined in later chapters). Stack behaviour is equivalent to the stack transition diagram. Possible values for *Stack-Behaviour* are shown in tables A.1 and A.2 of the same appendix.

1.3.3 Verifiability

Memory safety is a property that ensures programs running in the same address space are completely isolated from each other. Verifiability is a more strict property. Every program that is verified is memory safe, but some non-verifiable programs exist that are still memory safe.

Correctness is a property that ensures the execution of CIL code on all implementations of the CLI, with defined behaviour as specified in the standard. Correct CIL is not always verifiable. In that case, it is up to the compiler designer to ensure memory safety. For example, pointer arithmetic is not verifiable but it is essential in some cases. The application programmer must be careful in that case.

Correct CIL contains a **verifiable** subset. The verification algorithm shall attempt to associate a valid stack state with every CIL instruction. The stack state specifies the number of slots on the CIL stack at that point in the code and for each slot a required type that shall be present in that slot. The initial stack state is empty(there are no items on the stack). The verification algorithm shall simulate all possible control flow paths through the code and ensure that a valid stack state exists for every reachable CIL instruction. The verification algorithm does not take advantage of any data values during its simulation (e.g., it does not perform constant propagation), but uses only type assignments.

A useful attribute in CIL methods is **maxstack**. This value indicates the maximum number of items pushed that can be pushed onto the evaluation stack. Note that this number doesn’t specify the size of the stack in bytes at runtime, but rather the number of elements that will be tracked by an analysis tool. The information provided by *maxstack* is useful for *’CIL to native code’* compilers and verification.

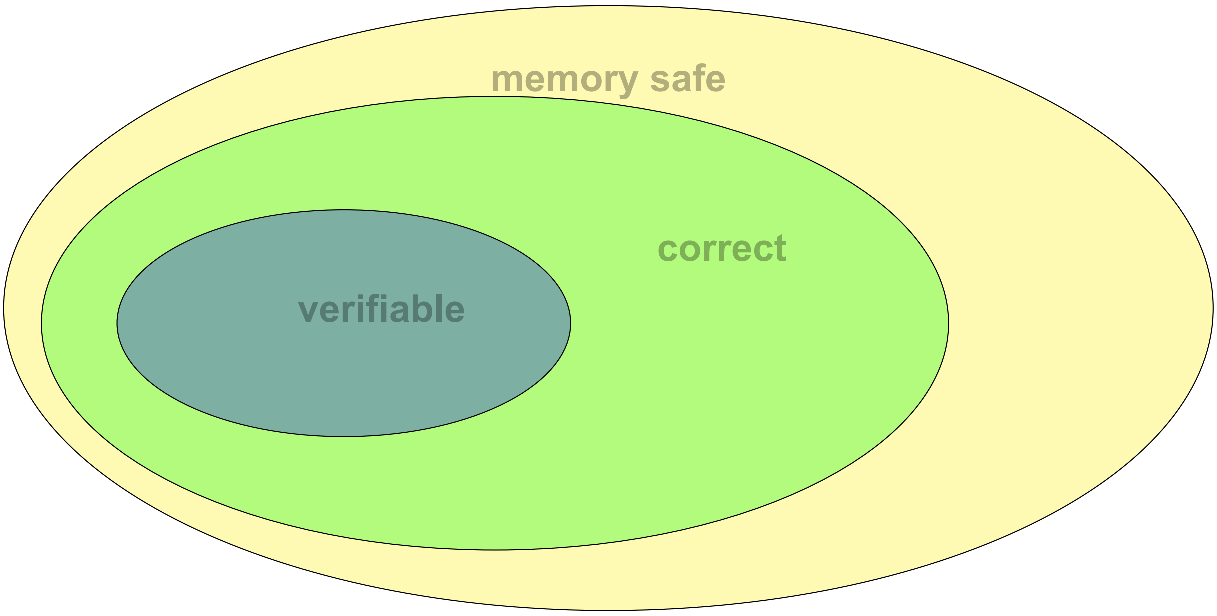


Figure 1.8: Venn diagram illustrating relationship between memory safe, correct and verifiable programs.

```
public static bool IsOdd(int x)
{
  return x % 2 == 1;
}
```

(a) C# 'IsOdd' method

```
 1 .method public static hidebysig default
 2     bool IsOdd (int32 x) cil managed
 3 {
 4 .maxstack 8
 5 IL_0000:  ldarg.0
 6 IL_0001:  ldc.i4.2
 7 IL_0002:  rem
 8 IL_0003:  ldc.i4.1
 9 IL_0004:  ceq
10 IL_0006:  ret
11 }
```

(b) CIL 'IsOdd' method

Figure 1.9: Comparison of C# and CIL methods that check if an integer is odd

1.3.4 Instruction Examples

To better understand the mechanism of code execution in the CLI we will look at 2 examples. In both examples the CIL code appears next to the corresponding high-level language code. For this purpose C#, an object oriented language, was chosen. A C# compiler, compiles the source code to a portable executable file format(PE) which ultimately contains the CIL code in binary format. At this point we can't do much with a binary file: the instructions are not readable by a human. We shall use a tool to convert the binary file to a text representation. The *Mono project* provides such a tool called **monodis** which is a disassembler for PE files.

Figure 1.9 illustrates a C# static method that takes a single integer argument and returns true if the argument is odd. When we look at the intermediate language method definition we see a lot of attributes at the start. The method attribute *'hidebysig'* stands for 'hide by signature' and it specifies that the declared method hides all methods of the base class types that have a matching method signature; when omitted, the method should hide all methods of the same name, regardless of the signature. The *'default'* attribute at line 2 means that the type *bool* has a defined default value. The 'cil managed' attribute pair

```
1 int Factorial(int n)
2 {
3   if (n == 0)
4     return 1;
5   return n * Factorial(n-1);
6 }
```

(a) C# 'Factorial' method

```
 1 .method private hidebysig instance default
 2     int32 Factorial (int32 n) cil managed
 3 {
 4 .maxstack 8
 5 IL_0000:  ldarg.1
 6 IL_0001:  brtrue IL_0008
 7 IL_0006:  ldc.i4.1
 8 IL_0007:  ret
 9 IL_0008:  ldarg.1
10 IL_0009:  ldarg.0
11 IL_000a:  ldarg.1
12 IL_000b:  ldc.i4.1
13 IL_000c:  sub
14 IL_000d:  call instance int32 class sample_cil
       .common_constructs::Factorial(int32)
15 IL_0012:  mul
16 IL_0013:  ret
17 }
```

(b) CIL 'Factorial' method

Figure 1.10: Comparison of C# and CIL factorial methods. At line 14 of CIL, "sample_cil" is the namespace of the method and "common_constructs" is the class.

states that the method contains standard managed CIL code. Now we can focus on the actual instruction stream.

The *.maxstack* directive specifies the maximum number of items pushed onto the evaluation stack as mentioned in section 1.3.3. Each instruction is preceded by an instruction label of the form *IL_xxxx*, which is a zero-indexed address. The first instruction at line 6, *ldarg.0*, loads the method argument at position 0 onto the evaluation stack(in this case x). The next instruction, *ldc.i4.2*, loads a constant value 2 on the stack. The 'i4' means that the constant 2 is a 4 byte integer. At line 8, the *rem* instruction calculates the remainder of the division of the last 2 elements of the evaluation stack. Until now, the corresponding calculated expression in the C# code is $x\%2$. Then the *ldc.i4.1* instruction loads the constant 1 onto the stack and the next instruction, *ceq*, pushes a 1 on the stack if the last 2 elements of the stack are equal or a 0 otherwise. This completes the desired calculation and the result is returned through the *ret* instruction, leaving the stack empty.

Another more complex example is shown in figure 1.10. The method 'Factorial' takes one integer argument n and recursively calculates the factorial of n. The *brtrue* instruction has an inline operand, which is the jump address in this case *IL_0008*. **brtrue** consumes the top element of the stack and based on its value, control is transferred either to the next instruction or to the branch target. We can infer this through the stack transition diagram too, as shown below:

$$..., value1 \quad \longrightarrow \quad ...$$

When this control flow instruction is executed and the value of the method argument is 0, lines 6-7 will load the constant 1 and return from the method, stopping the execution of the rest of the instructions. If that's not the case, control will get transferred to the instruction at line 9, and execution will continue from there. Notice how the state of the evaluation stack is identical, regardless of the control flow path that reaches the instruction at line 9.

Another interesting instruction is the one at line 14. We can easily see that before the

call instruction the stack contains 3 elements. The last added element is the result of subtracting 1 from n(*sub* instruction at line 13). The next element is the method argument 0. Unlike figure 1.9, where the method was static, here we have an instance method. This means that the first argument is always the instance object of the current class(**this** keyword). The object method argument is loaded onto the evaluation stack because it is needed by the call to the *Factorial* method at line 14. The stack transition diagram for *call* is the following:

$$..., \, arg0, \, arg1 \, ... \, argN \quad \longrightarrow \quad ..., \, retVal \text{ (not always returned)}$$

Finally, at line 9, the 1st argument(n) is loaded, in order to perform the multiplication needed for the calculation of the factorial(*mul* instruction at line 15).

2. INSPECTING MANAGED ASSEMBLIES

In the previous chapter we explained the basic aspects of the Common Language Infrastructure. We understood how source languages like C# are converted to CIL code. We presented the execution of several CIL programs by examining the evaluation stack carefully. Also we tried to describe each CIL instruction through concepts like the instruction variant table, stack transition diagram and verifiability/correctness conditions.

But how do we access all this information programmatically, given a PE file? This question is of great interest for our purposes: the ultimate goal is to **perform a conversion** on the instruction stream of CIL programs. We need a detailed representation of managed assemblies.

2.1 Assemblies, Modules, Classes and Methods

Assemblies and modules are grouping constructs, each playing a different role in the CLI. Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for CLI-based applications.

A **Module** is a logical collection of code within an Assembly. You can have multiple modules inside an Assembly, and each module can be written in different CLI-compliant languages(C#, F#, VB ...). Assemblies contain modules. Modules contain classes. Classes contain methods. Figure 2.1 depicts this relationship.

All this information about logical grouping of resources and code is called **Metadata**. It is defined in the files that contain the CIL code. The semantics of metadata, which dictate much of the operation of the VES, are described using the syntax of **ILAsm**, an assembly language for CIL(the same syntax illustrated in figures 1.9, 1.10, also referred as 'CIL code').

2.2 The Mono.Cecil Library

In this section the Mono.Cecil library is examined. We focus on the parts of the library that are necessary for our analysis. Cecil is a tool that enables us to **inspect and generate programs** and libraries in the ECMA CIL format. It provides the functionality to browse all the contained types of an assembly module, modify them on the fly and save back to the disk the modified assembly [2]. In our case, modifying programs is not needed.

Another similar .NET library for CIL generation and inspection is *System.Reflection.Emit*. There are two major differences that drive us into using *Cecil* for our purposes:

- Cecil has support for extracting the CIL bytecodes(parsing the binary files).
- Cecil does not need to load the assembly or have compatible assemblies to introspect the images.

The first thing to do when trying to extract the instructions of a program, is to load the module that contains the desired program. Let's look at an example that loads a module and prints the associated types of that module. The executable module is generated by

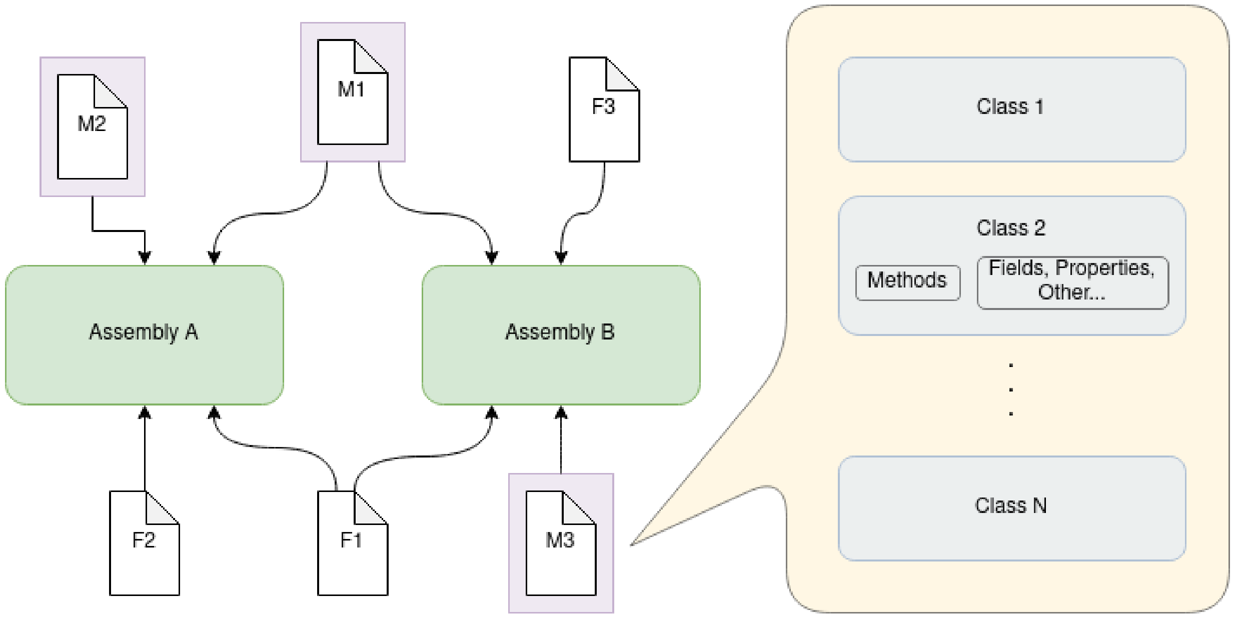


Figure 2.1: Relationship between assemblies and modules. Labels M1, M2 and M3 mark modules, whereas labels F1, F2, F3 mark resource files(configurations, images etc...). The contents of a module are illustrated on the right.

compiling C# code with the **mcs** compiler of *Mono*(by executing the following command: $ mcs HelloWorld.cs).

From here on, C# is used for all the analysis implementations. Figure 2.2 shows a 'Hello-World' program and illustrates the way we can browse through the types of that program.

2.3 Inspecting Instruction Properties

Now that we know how to browse the types of a module programmatically, we can continue on extracting the instruction stream of a type method. For example, if we want to print the OpCodes of each instruction of the Main method in figure 2.2 we could write code similar to that in figure 2.3. At line 5 and 6 of the same figure we use the LINQ (Language Integrated Query) extensions on *IEnumerable* objects.

Instruction objects of the Cecil library can give extensive information about their behaviour in CIL code. For example they provide information about the next and previous instructions, the OpCode and operand(s) of the instruction as well as the offset in the current method's instruction stream. A part of the *Instruction* class interface is shown in figure 2.4. The same figure illustrates the Opcode struct. For each instruction of a method we can obtain the corresponding *Mono.Cecil.Cil.Instruction* Object and from there we can navigate to the *OpCode*. OpCode objects can inform us on the *Name* of an instruction. For example the instruction at line 8 of figure 1.10, would have the name 'ret'.

The *FlowControl* property provides very useful information for our analysis, it specifies the kind of control flow behaviour of an instruction. An instruction that does not transfer control in the program has a special *FlowControl* value named 'Next'. This means that the instruction that will follow execution is the next instruction in the instruction stream. All possible *FlowControl* values are enumerated in table 2.1.

Given this representation we need not worry about the behaviour of every instruction of the instruction set, regarding control flow. Any implicit check on specific instructions is obsol-

```
class Hello
{
    public static void Main (string[] args)
    {
        System.Console.Out.WriteLine("Hello");
    }
}

class World { }
```

(a) When compiled, the single module will contain two classes.

```
static void PrintModuleTypes()
{
    ModuleDefinition module = ModuleDefinition
        .ReadModule("../sample-cil/HelloWorld.exe");
    foreach (var type in module.Types)
    {
        Console.Out.WriteLine(type.FullName);
    }
}
```

(b) The ’ModuleDefinition’ object is a representation of a module, provided by *Cecil*.

Figure 2.2: Illustration of browsing through types of a module. The code at (b) will output the name of the two types defined in (a)

```
static void PrintMethodInstructions() {
    ModuleDefinition module = ModuleDefinition
        .ReadModule("../sample-cil/HelloWorld.exe");
    var methodDefinition = module.Types
        .First(x => x.Name == "Hello").Methods
        .First(x => x.Name == "Main");
    foreach (var instruction in methodDefinition.Body.Instructions)
    {
        Console.WriteLine(instruction.OpCode.Name);
    }
}
```

Figure 2.3: Printing instruction OpCodes of a given method.

```
public sealed class Instruction
{
    public Instruction Next { get; set; }
    public object Operand { get; set; }
    public OpCode OpCode { get; set; }
    public int Offset { get; set; }
    public Instruction Previous { get; set; }
    public int GetSize();
    public override string ToString();
//...
}
```

(a) Members of the *Instruction* class.

```
public struct OpCode : IEquatable<OpCode>
{
//...
    public OperandType OperandType { get; }
    public OpCodeType OpCodeType { get; }
    public FlowControl FlowControl { get; }
//...
    public StackBehaviour StackBehaviourPop { get; }
    public StackBehaviour StackBehaviourPush { get; }
    public string Name { get; }
    public override string ToString();
//...
}
```

(b) Members of the *OpCode* struct.

Figure 2.4: *Instruction* and *OpCode* types.

ete. For example, instead of checking whether the name of an instruction is ’beq’ (branch on equal) and then from there inferring the behaviour implicitly, we can navigate into the instruction *OpCode* and then *FlowControl*. We will obtain such information frequently in the analysis of the next chapter.

Another matter to consider is obtaining branch targets. Branch target(s) of an instruction is the address(es) of all instructions within the same method that might be executed following it, excluding fall through (next). How do we obtain these target locations progammatically?

```
IL_0001: brtrue IL_0008
```

Consider the above example. When the value at the top of the evaluation stack is 1 (true), control gets transferred to *IL_0008*. We can obtain the instruction at *IL_0008* by accessing the current instruction’s *Operand*. Due to the fact that control flow targets do not receive their arguments solely from the evaluation stack (targets are constants in the code and cannot lie on the evaluation stack), they need to have operands. *Cecil* provides operands as objects, thus a cast is required in this case to obtain the target instruction object. Figure 2.5 shows a useful implementation that returns the targets of a control flow instruction.

Note that only in the case of a ’throw’ or ’ret’ instruction we return an empty array, because these instruction don’t have targets, in contrast they just stop execution of the method. Also, the only case in which the targets are more than one is that of the ’switch’ instruction, where all targets are inline to the instruction (encoded as multiple operands).

Table 2.1: Values for the *FlowControl* property of *OpCode* objects.

Branch	Unconditional branch, jumps to a specific target instruction
Break	Breakpoint, inform an available debugger to transfer control
Call	Method Call
Cond_Branch	Conditional Branch, based on a condition continue normal control flow or jump to a target
Meta	Unused operation or prefix code
Next	Continues normal control flow to the next instruction
Phi	Not used
Return	Returns control from the current method to the calling method
Throw	Transfers control to an available exception handler or propagates the exception up the call stack

```
public static IEnumerable<Instruction>
GetControlFlowInstructionTargets(this Instruction i)
{
    if (i.OpCode.FlowControl == FlowControl.Return
        || i.OpCode.FlowControl == FlowControl.Throw)
        return new Instruction[] { };
    else if (i.Operand is Instruction)
        return new Instruction[] { (Instruction)i.Operand };
    else if (i.Operand is Instruction[]) // This case is for the switch instruction.
        return i.Operand as Instruction[];
    throw new InvalidOperationException(
        @"The given instruction is not a control transfer instruction");
}
```

Figure 2.5: Method returning the targets of a control flow instruction. Note that a cast is required because *Operand* is an *Object*.

3. CONTROL FLOW GRAPH

Having explained how we can extract analysis information from method instructions, we can continue on developing the algorithms necessary for the conversion of stack-based intermediate representation to a linear intermediate representation (IR). The term ***Linear IR*** is used interchangeably with ***register-based IR***. The 'linear' part of the term dictates that each instruction specifies its arguments explicitly and the data on which it performs operations are entirely represented by arguments. Thus, when reading code, we don't need to keep context of what the previous instruction pushed on the stack (as seen for stack-based models), or what is the result being pushed on the stack. In contrast, we keep track of a register file, and read each instruction independently in a linear/forward-pass fashion. The term *LinearIR* is encountered extensively in *Mono* too, as an intermediate representation for the CLI implementation.

In this chapter, we will attempt to build the register-based representation by simulating the execution of the instruction stream on every single control flow path. When simulating execution, we can associate the contents of the stack with virtual registers and then extract a new representation for the instructions. In order to perform this simulation of control flow paths, we will need a control flow graph representation of a method. We will not address control flow across method calls though. This indicates that, for the purposes of the conversion, method calls are considered to have a *fallthrough* control flow behaviour.

3.1 Generally on CFGs

In computer science, a **control-flow graph** (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. This representation is essential to many compiler optimizations and static-analysis tools.

In a control-flow graph each **node** in the graph represents a **basic block**. A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit. This follows that code within a basic block is executed sequentially. Directed **edges** are used to represent jumps in the control flow. For example if control can flow from basic block A to basic block B, then a directed edge $A \rightarrow B$ exists in the CFG. There are two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. For our purposes, a method, can have more than one exit blocks, but always has one entry block.

Because of its construction procedure, in a CFG, every edge $A \rightarrow B$ has the property that:

$$Outdegree(A) > 1 \quad \text{or} \quad Indegree(B) > 1$$

Where $Outdegree(A)$ is the number of outgoing edges from A and symetrically, $Indegree(B)$ is the number of incoming edges to node B.

The CFG can thus be obtained, at least conceptually, by starting from the program's (full) flow graph (i.e. the graph in which every node represents an individual instruction) and performing an edge contraction for every edge that falsifies the predicate above, i.e. contracting every edge whose source has a single exit and whose destination has a single entry. This algorithm is not used in practice though, due to its increased complexity. In the next section we will describe an efficient implementation for CFG generation.

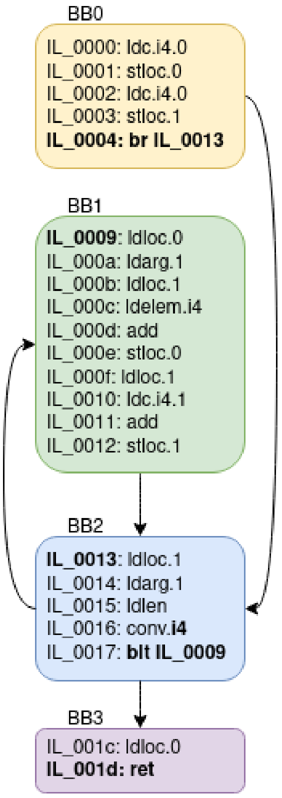
Let's consider an example of a control flow graph. Figure 3.1 illustrates a for loop that calculates the sum of array elements. We observe that the method is broken down to four basic blocks labeled 0 to 3. BB_0 is the entry block, whereas BB_3 is the exit block. The arrows depict the edges of the graph and the nodes contain the sequential code. Jump targets are shown in bold. It is clear how the for loop is working, when looking at the graph. We observe that blocks BB_1 and BB_2 form a cycle in the graph (edges $BB_1 \rightarrow BB_2$ and $BB_2 \rightarrow BB_1$).

```
int Sum(int[] A)
{
  int s = 0;

  for (int i = 0; i < A.Length; i++)
  {
    s+=A[i];
  }

  return s;
}
```

(a) *Sum* method

```
IL_0000:  ldc.i4.0
IL_0001:  stloc.0
IL_0002:  ldc.i4.0
IL_0003:  stloc.1
IL_0004:  br IL_0013
IL_0009:  ldloc.0
IL_000a:  ldarg.1
IL_000b:  ldloc.1
IL_000c:  ldelem.i4
IL_000d:  add
IL_000e:  stloc.0
IL_000f:  ldloc.1
IL_0010:  ldc.i4.1
IL_0011:  add
IL_0012:  stloc.1
IL_0013:  ldloc.1
IL_0014:  ldarg.1
IL_0015:  ldlen
IL_0016:  conv.i4
IL_0017:  blt IL_0009
IL_001c:  ldloc.0
IL_001d:  ret
```

(b) Method instructions.



(c) Corresponding CFG

Figure 3.1: A Method calculating the sum of an array with a for loop on the left. The corresponding instruction stream of the method is in the middle. The extracted CFG representation of the method is shown on the right.

Similar programming constructs like the while loop, the switch statement, if statements can be expressed via control flow graphs. A useful application of CFGs is that of determining *reachability*. If a subgraph is not connected from the subgraph containing the entry block, that subgraph is unreachable during any execution, and so is unreachable code; under normal conditions it can be safely removed. A common case of unreachable code occurs when a branch condition is known to evaluate to true or false at compile time. In this case, one of the control flow paths will never execute.

The next section, describes the algorithm used for the generation of a method's CFG.

3.2 Generating a GFG representation

A control flow graph models all program executions as we saw earlier. The basic idea behind CFG generation is that once execution enters a basic block, all instructions inside

```
    add    $t0, $gp, $zero          # &A[0] - 28
    lw     $t1, 4($gp)          # fetch N
    sll    $t1, $t1, 2          # N as byte offset
    add    $t1, $t1, $gp        # &A[N] - 28
    ori    $t2, $zero, 256      # MAX_SIZE
top:
    sltu   $t3, $t0, $t1        # have we reached the final address?
    beq    $t3, $zero, done     # yes, we're done
    sw     $t2, 28($t0)         # A[i] = 0
    addi   $t0, $t0, 4          # update $t0 to point to next element
    j      top                  # go to top of loop
done:
```

Figure 3.2: MIPS Assembly for a *for loop*. We observe that unlike CIL code, branch targets are specified through labels. The label *top* corresponds to the top of the loop and label *done* is reached when the loop ends.

the basic block are executed no matter what. In other words it is a single-entry/single-exit region of code.

Building the CFG essentially requires two main operations [7]:

1. Group instructions into basic blocks
2. Link basic blocks by analyzing jumps

In our case though, we will need to add a pre-processing step because jump targets are not known in advance and shall be inferred by a pass on the instruction stream.

We implement a class named *CilControlFlowGraph*, which is the representation of CFG. Objects of this class are constructed based on a *MethodDefinition* object that contains the instruction stream. When constructed, these objects provide us with useful methods that expose the CFG. The graph is encoded as a set of nodes that contain navigation information that represent edges. Nodes are encoded as *CilBasicBlock* objects. From a basic block object we can access the fallthrough, in and out basic blocks, its instructions and string representations. More on the design of the representations can be found on the *Doxygen* documentation.

3.2.1 Finding Branch Targets

Identifying branch targets is a process where we discover ’labels’ of the code. A label in conventional assembly languages is an instruction address in some arbitrary code. Labels are used for specifying jump targets. Instead of displaying the address of each instruction, assembly code sometimes displays only the addresses of instructions that are jump targets. Also instead of writing complicated addresses we can write down a simple label string like ’L0’, ’L3’ or even meaningful phrases like ’exit’ or ’is_odd’, indicating some control flow path. An example of such code is shown in figure 3.2.

In our analysis, when forward passing the instruction stream we don’t know if the current instruction is a branch target in advance. Note that *Cecil* provides branch targets as properties of a branch instruction. Based on these observations, we can perform a pass on the method instructions, in order to save the branch targets in an appropriate data structure. We do this by using the branch target extension methods presented in figure 2.5 of a

```
1 private void FindBranchTargetInstructions()
2 {
3     branchTargetInstructionDictionary = instructions
4         .Where(x => x.IsControlFlowInstruction())
5         .SelectMany(x => x.GetControlFlowInstructionTargets())
6         .Distinct() // leave instructions have the same target.
7         .ToDictionary(k => k, v => (CilBasicBlock)null);
8 }
```

Figure 3.3: Implementation of branch target identification

previous section. We store the branch target information in a map (dictionary). This map corresponds to pairs of branch target instructions and basic blocks and will be used later to link basic blocks. Figure 3.3 shows the implementation of this pass on the CIL.

Note that there exists a chance that the flattening at line 5 produces duplicate items in the resulting collection. As a result we need to apply the *Distinct()* operation. These ramifications are due to the instructions **leave** and **leave.s**. These instruction are used in exception handlers as the exit point from **multiple catch/filter blocks** and therefore have the same target. This common target, is the instruction immediately following the try-catch construct. We will consider exception handling more carefully in later sections.

3.2.2 Constructing Basic Blocks

The previous section was a pre-processing step for the construction of basic blocks. We now have access to all branch targets via a dictionary's keys (*branchTargetInstructionDictionary*). Before starting to group instructions into basic blocks, we need to consider what are the properties of starting and ending basic block instructions.

A first logical observation is that a basic block starts immediately after a conditional or unconditional branch instruction. This is obviously true by definition of the basic block. Symmetrically, a basic block ends at a branch instruction. Although these two criteria seem to be enough in order to generate CFG nodes (basic blocks), they don't describe some cases of basic block border instructions. For example, consider the instruction *IL_0012* of figure 3.1. This instruction is not a branch instruction (*stloc.1*), but it signals the end of BB_1. One thing that we notice though in this case, is that the next instruction of BB_2, *IL_0013*, is a **branch target**. This means that control can flow into this instruction from two different sources: the previous instruction (fallthrough) or the instruction that branches to this instruction.

Generalizing this idea, we can conclude that a basic block starts at a branch target. Symmetrically, just like before, we can easily infer that a basic block ends before a branch target instruction. In conclusion, we obtain the following 4 straightforward rules for basic block borders:

Basic blocks **start**:

- At a branch target
- After a conditional or unconditional branch

Basic blocks **end**:

```
 1 private void ConstructBasicBlocks()
 2 {
 3   int basicBlockId = 0;
 4   var currentBasicBlock = new CilBasicBlock(basicBlockId);
 5   foreach (var instruction in instructions)
 6   {
 7     currentBasicBlock.Instructions.Add(instruction);
 8     // Update the dictionary for the branch target instructions.
 9     if (branchTargetInstructionDictionary.ContainsKey(instruction))
10     {
11       branchTargetInstructionDictionary[instruction] = currentBasicBlock;
12     }
13     if (instruction.Next == null // Next BB starts at branch target.
14         || branchTargetInstructionDictionary.ContainsKey(instruction.Next)
15         || instruction.IsControlFlowInstruction()) // BB ends at branch.
16     {
17       // When we reach the end of the Basic block
18       // we add it to the result list.
19       BasicBlocks.Add(currentBasicBlock);
20       currentBasicBlock = new CilBasicBlock(++basicBlockId);
21     }
22   }
23 }
```

Figure 3.4: Implementation of basic block construction using the *branchTargetInstructionDictionary*

- At a conditional or unconditional branch
- Before a Branch target

Notice how it is essential to know information about branch targets in order to identify basic blocks. After discovering these useful properties of basic blocks, we can proceed to implement a simple algorithm that performs a pass on method instructions and creates a list of basic block objects, physically ordered as they appear in the code. Figure 3.4 illustrates method *ConstructBasicBlocks*, which uses the previously identified branch targets in order to construct basic blocks.

At this point, one may guess the purpose of *branchTargetInstructionDictionary*. As we see on lines 8 to 12 of figure 3.4, the current basic block is mapped to the branch target instruction, if any. When linking basic blocks we can infer the linked basic block from the branch target of a control flow instruction. The next section explains this in more detail.

3.2.3 Linking Basic Blocks

The final step of CFG generation is the addition of edges to the graph nodes. We model edges as navigational properties of basic blocks as described previously. *InBasicBlocks* is the list of incoming edges of a node and *OutBasicBlocks* is the list of outgoing edges. Regarding the conversion from stack-based to register-based representation we will not need information about in-basic-blocks. Nonetheless, we include it in the CFG for a more generic representation which surely can be used in a backwards analysis.

The key idea of this pass, is that we iterate through every basic block and set its *OutBasicBlocks* as well as the *InBasicBlocks* of every outgoing basic block, in a symmetric

```
 1 private void LinkBasicBlocks()
 2 {
 3   for (int i = 0; i < BasicBlocks.Count; ++i)
 4   {
 5     var currentBasicBlock = BasicBlocks[i];
 6     var lastInstruction = currentBasicBlock.LastInstruction;
 7     if (lastInstruction.IsControlFlowInstruction())
 8     {
 9       var targetInstructions =
10         lastInstruction.GetControlFlowInstructionTargets();
11       var targetBasicBlocks = targetInstructions
12         .Select(FindInstructionBasicBlock)
13         .ToList();
14       targetBasicBlocks.ForEach(currentBasicBlock.OutBasicBlocks.Add);
15       targetBasicBlocks.ForEach(x => x.InBasicBlocks.Add(currentBasicBlock));
16     }
17     if (lastInstruction.HasFallthroughInstruction())
18     {
19       // Fall through.
20       var nextBasicBlock = BasicBlocks[i + 1];
21       // The last basic block is always the fallthrough.
22       currentBasicBlock.OutBasicBlocks.Add(nextBasicBlock);
23       nextBasicBlock.InBasicBlocks.Add(currentBasicBlock);
24     }
25   }
26 }
```

Figure 3.5: Linking basic blocks with a simple pass on the basic block list.

fashion. This approach ensures that all basic blocks have their in/out basic blocks updated. Figure 3.5 shows a possible implementation of this algorithm.

Let's consider the body of the for loop in the same figure. First, we obtain the last instruction of the current basic block. We need to identify this instruction: is it a control flow instruction? If that's the case then we need to find the branch targets and from there navigate the out basic blocks via our branch target map. We link the basic blocks symmetrically as described previously. After this step, we should also consider fallthrough. Lines 17 to 26 check whether a fallthrough exists. Note that the condition expression at line 17 evaluates to true if the last instruction has a fallthrough (as a means of control flow behaviour). The body of the if statement will never execute for an exit basic block because the last instruction of an exit basic block always has a *FlowControl* property of *Throw* or *Ret*. Otherwise it is not an exit block regarding our analysis. Therefore the expression $BasicBlocks[i+1]$ in the last if-statement is safe to use, when i enumerates all basic blocks.

In conclusion, we have successfully generated a CFG representation of an arbitrary method. 3 passes were performed. The first and second are passes on the instruction stream with $O(I)$ time and memory complexity, where I is the number of instructions. We assume that the number of edges, or equivalently the number of branch targets, is constant and small (1 or 0 most of the time, aside from fallthrough). The third pass is performed on the basic blocks and has $O(E)$ time complexity, where E denotes the number of edges in the graph. Note that, $E = O(B)$, where B is the number of basic blocks. This means that the complexity of linking basic blocks is linear to the number of basic blocks. Overall, we observe that the complexity of CFG generation is $O(I+B)$ and because $B << I$, the final result is $O(I)$: linear to the number of instructions.

3.3 Exception Handling and CFG

Object oriented code often supports exception handling through some language constructs called **exception handlers**. The programmer specifies a region of code that may throw an exception and then writes some code that handles the exception if it occurs at runtime. Although this idea may seem simple at the beginning, it involves many implementation details and verifiability conditions regarding CIL code.

Let's see an example of a method containing a try-catch block, its corresponding intermediate code and the produced control flow graph. This will give us an insight on the properties of exception handlers.

```
static int TryCatch(int x) {
  try {
    return 1/x;
  } catch (
    DivideByZeroException) {
    return -1;
  } catch (Exception) {
    return -2;
  }
}
```

(a) Try-catch method

```
.try {
  IL_0000:  ldc.i4.1
  IL_0001:  ldarg.0
  IL_0002:  div
  IL_0003:  stloc.0
  IL_0004:  leave IL_001a
}
catch class [mscorlib]System.
    DivideByZeroException {
  IL_0009:  pop
  IL_000a:  ldc.i4.m1
  IL_000b:  stloc.0
  IL_000c:  leave IL_001a
}
catch class [mscorlib]System.Exception {
  IL_0011:  pop
  IL_0012:  ldc.i4.s 0xfffffffe
  IL_0014:  stloc.0
  IL_0015:  leave IL_001a
}
IL_001a:  ldloc.0
IL_001b:  ret
```

(b) Method instructions.

Figure 3.6: Illustration of exception handlers with CFG.

Figures 3.6 and 3.7 show the code and CFG for a method that performs division by an argument x and catches a *DivideByZeroException* or a base exception. The code is divided into 4 basic blocks BB_0, BB_1, BB_2 and BB_3. The first basic block corresponds to

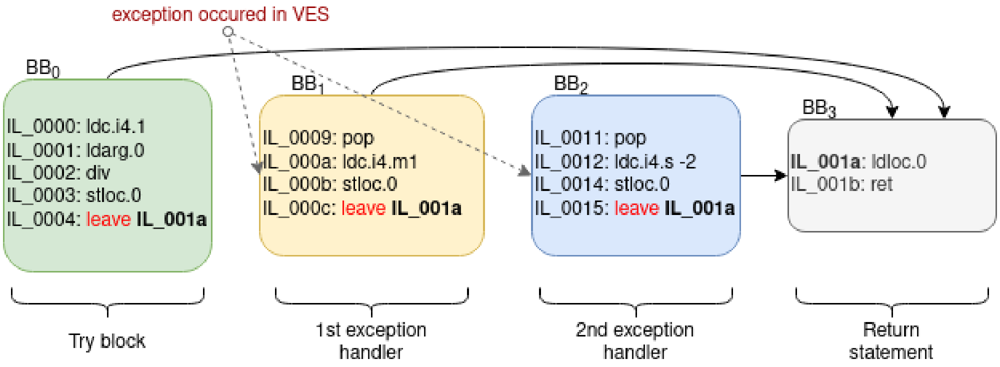


Figure 3.7: Control flow graph for a try catch construct with 2 handlers. An exception handler has no entry point, because it does not have to do with normal program execution.

the try-block. It ends with the instruction **leave** and the target of the instruction is BB_3. The second basic block corresponds to the exception handler that catches *DivideByZeroException* exceptions. The third block is associated with the last handler which catches any exception. These blocks end in a *leave* instruction too, with the same target as the first one.

It is worth noting that the *.try, catch and '{' '}'* are not part of the actual code in figure 3.6. They are just printed for ease of reading the code. The information needed to identify handlers and try blocks is encoded as start-end addresses of the related instructions. For example, for BB_0 the bytecode would contain somewhere the following information:

$$\text{tryBlock}: \qquad startAddress = IL\_0000 \quad endAddress = IL\_0009$$

Let's look at the instructions of a handler, i.e BB_1. The first instruction is **pop**. This instruction pop's an element from the evaluation stack as the name indicates. This is needed because when an exception occurs the VES pushes the exception object onto the stack. After the exception is popped, -1 is loaded in order to be returned eventually. Then, the leave instruction branches unconditionally to the last basic block. Notice how the properties of basic blocks still stand here. BB_0 **ends at a branch instruction** and **starts after a branch** (except, of course, of the first block). It is easy to check that the same holds for other basic blocks too.

An important difference of the CFG of figure 3.7 from that of previously shown CFGs is that not all nodes are **reachable**. In other words if we analyze forward jumps in the graph then we won't consider the handler nodes. In our case of code conversion, this is a problem, because we must convert all instructions of a method. We will address this problem in the next chapter with two solutions.

4. REGISTER BASED REPRESENTATION FROM CONTROL FLOW GRAPH

Stack based models rely on saving intermediate results on a stack as we saw. In contrast, register based architectures, save operands in registers. Conceptually, on a higher level, registers can be thought of as a set of variables or as an array of values. We are not interested in registers as hardware components. The naming remains, though, because in compiler terminology registers are associated with assembly code generation [3].

How do we transition from stack-based IR to register-based IR?

The key idea lies in simulating code execution, in terms of stack behaviour. When we refer to stack behaviour, we are interested in how many elements are pushed or popped on the evaluation stack with each instruction execution. It is possible to track types of elements too, but in this case it is unnecessary. By simulating this behaviour we can infer the state of the stack at any instruction, for any execution.

Once the above analysis is implemented, we can associate each slot of the stack with a register in a simple fashion. The one-to-one relation between stack slots and registers ensures that the **number of registers used** for an arbitrary method is equal to the **.maxstack** attribute of the method.

Following these observations, one can notice why a control flow graph can be useful. Simulating code execution requires traversing valid control flow paths. These paths are exactly described with a CFG. Execution paths can be simulated by performing a **depth-first traversal** on the graph. Of course, some modifications on the traditional DFS algorithm may be needed, in order to address reachability. We will discuss this in later sections.

4.1 Defining *LinearIr* Objects

The conversion to stack-based IR is modeled with *LinearIr* objects. As a reminder, the term linear IR is used in exchange with register-based IR. Initially we define a base abstract class that contains the utilities for evaluation stack behaviour that is common to any conversion algorithm. Then, different implementations can inherit from this class in order to apply their algorithm. Figure 4.1 shows the simple inheritance diagram for LinearIr.

The *LinearIr* class contains methods that provide the following:

- The method definition
- A stream of register-based instructions
- The number of registers used by the method

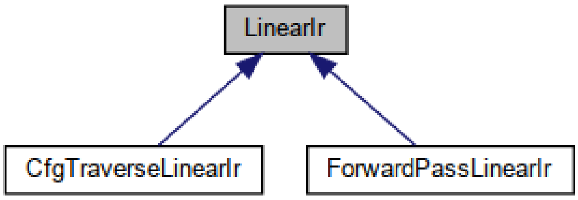


Figure 4.1: Inheritance diagram for *LinearIr*. *ForwardPassLinearIr* and *CfgTraverseLinearIr* are different implementations of linear IR.

- Printing utilities for the register-based instructions.

We observe that our goal here is to produce a sequence of linear IR instructions ordered by their physical order in the initial method instruction stream. Linear IR instructions are represented by *LinearIrInstruction* objects. These objects are constructed based on a stack-based instruction of the *Cecil* library. This ensures that a **one-to-one** relationship exists between representations. Also, this solves the problem of redefining common properties of stack-based and register-based instructions, like *OpCodes*, *Operands* etc. A linear IR instruction contains a list of **input** and **output registers**. These registers are set when performing the conversion algorithm.

4.2 Input/Output Instruction Registers

At this point our concern is how to decide how many (and which) registers, we should add at a given linear IR instruction. For example an *add* instruction obviously should have 2 input registers and a single output register. We should infer the following syntax from a stack-based *add* instruction:

```
IL_XXXX: v2 <- add v0 v1
```

where v0, v1 and v2 are logically correct registers, meaning that v0 and v1 contain some data to sum and v2 is used as the result.

In order to achieve this, we employ an approach that updates a variable called *evaluationStackSize*, which indicates the size of the evaluation stack at any point of a method. This approach is common to both algorithms(*ForwardPass* and *CfgTraverse*). Initially the stack size is set to 0 (at the start of a method). Then we iterate through instructions. For each instruction, we increase or decrease the stack size based on its stack behaviour. For example, in the case of *add*, the stack size is decremented. This is expected, because the *add* instruction consumes 2 items from the stack and pushes 1 item. Overall the stack size is reduced by 1 item.

This process is continued for every instruction. When we reach the end of the method, a *ret* instruction is present and after the execution of that instruction the stack should be empty. This assertion is used in unit tests to ensure that the execution simulation was successful.

To get a better idea of the execution simulation let's consider figure 4.2, which illustrates a case analysis for the pushing stack behaviour of an instruction. If the instruction doesn't push anything to the stack then no action is needed. In any other case, the number of pushed items is used to create output registers. At line 16 of the same figure, the stack size is used as an id for the new register. If, for example, at that point the stack size is 4, then the output register would be **v3** (indexed from 0).

The last case, which addresses the stack behaviour **Varpush**, is quite different. The number of elements pushed to the stack is not known in advance. With a short study of instruction properties, we observe that the only instructions that fall under the *Varpush* case are **call**, **callvirt** and **calli**. For these 3 instructions, the number of returned items is either 1 or 0, because a method call either returns something or has a *System.Void* return type and therefore does not return a value. We analyze the return type and take the corresponding action.

```
protected int[] GetInstructionOutputRegisters(Instruction i)
{
  int[] outputRegisters = null;
  switch (i.OpCode.StackBehaviourPush)
  {
    case StackBehaviour.Push0:
      outputRegisters = new int[0];
      break;
    case StackBehaviour.Push1:
    case StackBehaviour.Pushi:
    case StackBehaviour.Pushi8:
    case StackBehaviour.Pushr4:
    case StackBehaviour.Pushr8:
    case StackBehaviour.Pushref:
      outputRegisters = new int[1];
      outputRegisters[0] = evaluationStackSize++;
      break;
    case StackBehaviour.Push1_push1:
      outputRegisters = new int[2];
      outputRegisters[1] = evaluationStackSize++;
      outputRegisters[0] = evaluationStackSize++;
      break;
    case StackBehaviour.Varpush:
      // Instructions that have this behaviour: callvirt calli call
      // This is dealt simmilarly to the ret instruction of the current method
      var methodToCall = i.Operand as MethodReference;
      bool methodReturnTypeIsVoid =
        methodToCall.ReturnType.FullName == "System.Void";
      outputRegisters = new int[methodReturnTypeIsVoid ? 0 : 1];
      if (!methodReturnTypeIsVoid)
      {
        outputRegisters[0] = evaluationStackSize++;
      }
      break;
    default:
      throw new InvalidOperationException("Invalid Stack Behaviour");
  }
  // return the result we obtained by the case analysis.
  return outputRegisters;
}
```

Figure 4.2: Simulating items pushed on the stack when an instruction is executed. Output registers are computed based on stack behaviour. *GetInstructionInputRegisters* is implemented in a symmetric fashion with minor tweaks.

One assumption that was made, is the fact that a change in evaluation stack size is always reflected as instruction stack behaviour. This is not true though. When an exception occurs during normal program execution, the VES pushes an exception object onto the evaluation stack. This action is not described via stack behaviour, because obviously it is unexpected. To address this problem we analyze protected regions, as well as handlers and make sure that when entering an exception handler, the stack size is incremented to reflect the pushed exception object. This modification ensures the correctness of the execution simulation.

At this point we may introduce a new assembly string format for linear IR instructions. This will help us read the output of our conversion. The string representation must be straightforward. As we described previously, popped/pushed values from the stack are associated with input/output registers respectively. It is therefore a logical design, to include register information in the assembly format. Another useful design, would be to create a distinction between input and output registers. Last but not least, the instruction opcodes and addressing should be kept common between register-based and stack-based representations. Based on these thoughts, a proposed format for linear IR could be the following:

$$\text{IL\_XXXX}: \; outReg_0 \ldots outReg_n \; \leftarrow \; OpCode\ Operand(s) \quad inReg_0 \ldots inReg_m$$

The above format specifies all the information needed to understand the execution of a program. It is worth noting that, when there are no output registers ($n = 0$), the $\leftarrow$ is not printed. Similar modifications may apply to input registers as well as targets, because $n \geq 0$, $m \geq 0$ and $k \geq 0$, which means that these values can be 0 in many cases.

4.3 CFG Traversal

In the previous section, we studied how a stack-based instruction corresponds to a linear IR instruction. Moving a level upwards in the organization of code, we should decide how a stack-based basic block corresponds to a linear IR basic block. Fortunately, this correspondence is simple: just apply the individual instruction conversion to each instruction of the basic block.

The next logical step is to convert all basic blocks. This is not trivial though. As we described before, we are simulating code execution. Therefore, we can not iterate through basic blocks in an arbitrary order. For a given ordering of basic blocks, we can continue conversion from block BB_i to BB_{i+1} if and only if an edge between the former exists in the CFG. Otherwise, the state of the stack will be unknown between transition from one basic block to the next. Figure 4.3 illustrates this restriction.

At this point, our intuition suggests that we follow an approach that performs a valid traversal of the control flow graph. A breadth first traversal does not produce a traversal order with valid control flow. What we are looking for, is a **depth first** traversal of the control flow graph. The idea behind depth first search is that we follow a path until we reach an exit block and then we backtrack to discover other paths too. This procedure suits our needs, because it simulates execution of all control flow paths.

Figure 4.4 shows a **recursive** implementation of the depth-first traversal algorithm. The method is called with the entry basic block as its argument. The linear IR for the entry

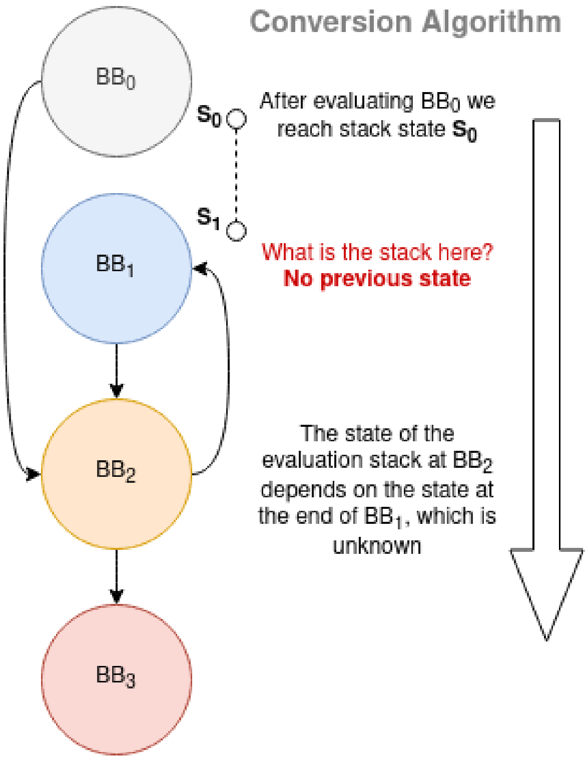

Conversion Algorithm
BB$_0$
S$_0$
After evaluating BB$_0$ we reach stack state **S$_0$**
S$_1$
BB$_1$
What is the stack here?
No previous state
BB$_2$
The state of the evaluation stack at BB$_2$ depends on the state at the end of BB$_1$, which is unknown
BB$_3$

Figure 4.3: Restriction on the evaluation of basic blocks with an arbitrary order. The stack state in not known in all basic block transitions.

```
1  // ...
2  private Dictionary<CilBasicBlock, IEnumerable<LinearIrInstruction>> visited;
3  // ...
4  private void RecursiveDFS(CilBasicBlock basicBlock)
5  {
6    visited.Add(basicBlock, GetBasicBlockLinearIrInstructions(basicBlock));
7    int evaluationStackSizeSnapshot = evaluationStackSize;
8    foreach (var outBasicBlock in basicBlock.OutBasicBlocks)
9    {
10     evaluationStackSize = evaluationStackSizeSnapshot;
11     if (!visited.ContainsKey(outBasicBlock))
12     {
13       RecursiveDFS(outBasicBlock);
14     }
15   }
16 }
```

Figure 4.4: DFS traversal implementation for a method's CFG. The search algorithm is used to simulate normal program execution. The variable *evaluationStackSizeSnapshot* is used to remember the stack size when entering a control flow path. This way, when the path is explored, other paths can be explored too starting with a valid stack state. The conversion is performed on line 6 by a call to method *GetBasicBlockLinearIrInstructions*.

block is computed and the block is marked as visited. A dictionary that maps basic blocks to their corresponding register based representation is updated for this purpose. At line 7, all the *out* basic blocks are examined. If an out basic block is visited, it is ignored. Otherwise, a snapshot of the stack size at that point is taken before the recursive call to *RecursiveDFS*. When the called method returns, the stack size is restored and the next out basic block is examined.

A problem arises when not all nodes of the CFG are reachable. We examined how this can happen in section 3.3. However, the solution to this problem is simple. Instead of starting the recursive method from the entry block, we iterate all basic blocks and apply the method to every node of the CFG that is not yet visited. This ensures that all nodes are visited. Notice that we still follow valid control flow paths. When converting basic blocks that have no entry block we make sure that the stack is empty. Note that all unreachable nodes, are associated with exception handling and the CLI standard ensures that in these cases the stack is always empty.

5. REGISTER BASED REPRESENTATION FROM FORWARD PASS

In the previous chapter, we presented an algorithm that made use of the control flow graph representation in order to perform the conversion. This algorithm was agnostic to any properties of control flow paths regarding the stack state. The only assumption that we made was about exception handlers starting with an empty stack, which is not that useful.

In this chapter we will try to perform the conversion from stack-based to register-based IR, without the help of a CFG. Instead, a forward pass on the instruction stream should be enough. In order to achieve this, we must explore the properties of CIL methods. Before moving further, let's consider why a forward pass conversion is not possible without constraints on the instruction stream.

5.1 Backward Branch Constraints

When reaching an instruction that immediately follows an unconditional branch and where that instruction is not some branch target of an earlier instruction, we can't infer the state of the stack. This was our main problem with the CFG algorithm too, where we solved it by simulating execution through the CFG. In this algorithm we are *trying* to simulate execution where it is possible. When reaching a branch, we continue converting even though the control flow path we chose is not a valid program execution. In order to justify this decision, we rely on the following **backward branch constraints**, as they appear in the CLI ECMA [4]:

It shall be possible, with a single forward-pass through the CIL instruction stream for any method, to infer the exact state of the evaluation stack at every instruction (where by 'state' we mean the number and type of each item on the evaluation stack).

In particular, if that single-pass analysis arrives at an instruction, call it location X, that immediately follows an unconditional branch, and where X is not the target of an earlier branch instruction, ***then the state of the evaluation stack at X, clearly, cannot be derived from existing information****. In this case,* ***the CLI demands that the evaluation stack at X be empty****.*

Following on from this rule, it would clearly be invalid CIL if a later branch instruction to X were to have a non-empty evaluation stack.

[Rationale: This constraint ensures that CIL code can be processed by a simple CIL-to-native-code compiler. It ensures that the state of the evaluation stack at the beginning of each CIL can be inferred from a single, forward-pass analysis of the instruction stream. end rationale]

[Note: the stack state at location X in the above can be inferred by various means: from a previous forward branch to X; because X marks the start of an exception handler, etc. end note]

```
private void SetLinearIrInstructions()
{
  var instructionArray = new LinearIrInstruction[MethodDefinition.Body.Instructions.Count];
  for (int i = 0; i < MethodDefinition.Body.Instructions.Count; i++)
  {
    var stackBasedIrInstruction = MethodDefinition.Body.Instructions[i];
    var linearIrInstruction = GetLinearIrInstructionFrom(stackBasedIrInstruction);
    instructionArray[i] = linearIrInstruction;
    if (stackBasedIrInstruction.IsControlFlowInstruction())
    {
      var targets = stackBasedIrInstruction.GetControlFlowInstructionTargets();
      foreach(var target in targets)
      {
        if (!stackSizeAtBranchTarget.ContainsKey(target))
          stackSizeAtBranchTarget.Add(target, evaluationStackSize);
      }
    }
    if (!stackBasedIrInstruction.HasFallthroughInstruction())
    {
      if (stackBasedIrInstruction.Next != null &&
          stackSizeAtBranchTarget.ContainsKey(stackBasedIrInstruction.Next))
        evaluationStackSize = stackSizeAtBranchTarget[stackBasedIrInstruction.Next];
      else
        evaluationStackSize = 0;
    }
  }
  Instructions = instructionArray;
}
```

Figure 5.1: Forward pass implementation. The register-based instructions are stored into an array. In each iteration, instructions are checked for control flow behaviour and the *stackSizeAtBranchTarget* dictionary is updated with new information on the IR.

5.2 Forward Pass Implementation

For this algorithm, we will be implementing class ForwardPassLinearIr which derives from LinearIr. As we described in the previous chapter, stack behaviour utilities are preserved and the process of converting one single instruction is kept the same. We introduce a new data structure, which will help us with branch targets. Throughout the conversion process a dictionary called *stackSizeAtBranchTarget* is updated. This dictionary, maps control flow instruction targets, to an integer indicating the size of the stack at the target.

This kind of information is essential to the algorithm, because when we encounter an instruction immediately following an unconditional branch, where the instruction is a target of a previous control flow instruction, we can infer the stack size at that point. A simple search in the previously defined dictionary will provide us with the desired stack state. The target is used as a key to index into the dictionary.

Figure 5.1 shows the implementation of the forward pass. Initially, an array with size equal to the instruction stream is created. Then a for loop iterates through the stack-based instructions. Each instruction is converted and assigned to its position in the array. As a side-effect, the conversion updates the evaluation stack size. Then, we check if the current instruction has any targets and we add them to the dictionary that we defined earlier. Finally, the instruction is checked for fallthrough, and if so, the evaluation stack size is reset to 0, or to a valid size found in the dictionary.

This approach, is superior to the CFG approach in terms of code complexity and efficiency. This is expected, because as the name indicates it is just a forward pass on the method. But we shall not forget that this method works because of the carefully designed architecture of CLI. Was it not for backwards branch constraints and we could not infer stack state at all cases.

6. CONCLUSIONS AND FUTURE WORK

Having described our solution to the problem of converting stack-based to register based IR, let's compare the two representations and highlight their similarities as well as their differences.

6.1 Representation Comparison

```
.method private static hidebysig
    default int32 Factorial (int32 n)
     cil managed
{
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: brtrue IL_0008
  IL_0006: ldc.i4.1
  IL_0007: ret
  IL_0008: ldarg.0
  IL_0009: ldarg.0
  IL_000a: ldc.i4.1
  IL_000b: sub
  IL_000c: call int32 class
    sample_cil.common_constructs::
    Factorial(int32)
  IL_0011: mul
  IL_0012: ret
}
```

```
System.Int32 sample_cil.
    common_constructs::Factorial(
    System.Int32)
{
  IL_0000: v0 <- ldarg.0
  IL_0001: brtrue IL_0008 v0
  IL_0006: v0 <- ldc.i4.1
  IL_0007: ret v0
  IL_0008: v0 <- ldarg.0
  IL_0009: v1 <- ldarg.0
  IL_000A: v2 <- ldc.i4.1
  IL_000B: v1 <- sub v1 v2
  IL_000C: v1 <- call System.Int32
    sample_cil.common_constructs::
    Factorial(System.Int32) v1
  IL_0011: v0 <- mul v0 v1
  IL_0012: ret v0
}
```

Figure 6.1: Comparison between stack-based and register-based representations.

Figure 6.1 illustrates the *Factorial* method that we encountered in previous chapters. On the left, we see the stack-based CIL assembly format and on the right, the linear IR that was generated. The first thing to notice, is that the sequence of instructions is unchanged across representations. Another observation is that the maximum number of items pushed to the stack is equal to the number of registers used (this is shown in figure 6.2). Also note that when a *ret* instruction occurs, its input register is always *v0*. This happens because when a method exits, the evaluation stack must be empty. Hence, register *v0* is mapped to the single element of the stack, before the *ret* instruction is executed.

Unlike the original assembly format, the linear IR is easier to read because we can track register names by looking at the code, without having to visualize a stack. Also, we have a more clear representation of information flow.

6.2 Performance

Another important aspect of the conversion analysis is performance. As we highlighted, when presenting the conversion algorithms, the complexity of our solution is linear to the number of method instructions. As a result, we should expect fast conversion times even for large code-bases. Another expectation is that the *ForwardPass* implementation will be most definitely faster than *CfgTraverse*.

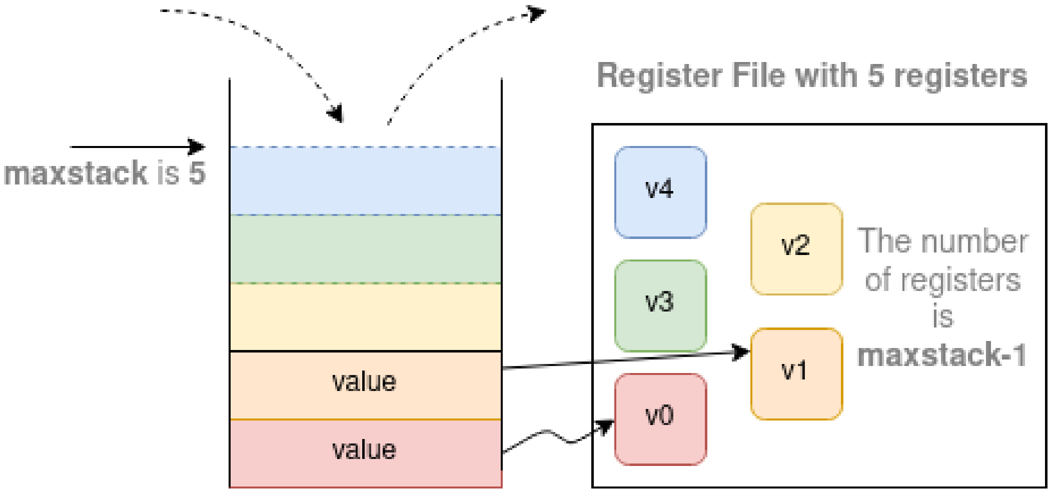

maxstack is 5
value
value
Register File with 5 registers
v4
v3
v2
v1
v0
The number of registers is maxstack-1

Figure 6.2: The 1-to-1 correspondence of stack slots and registers. Note that the bottom of the stack is always associated with register *v0*, the item above the bottom with register *v1* etc...

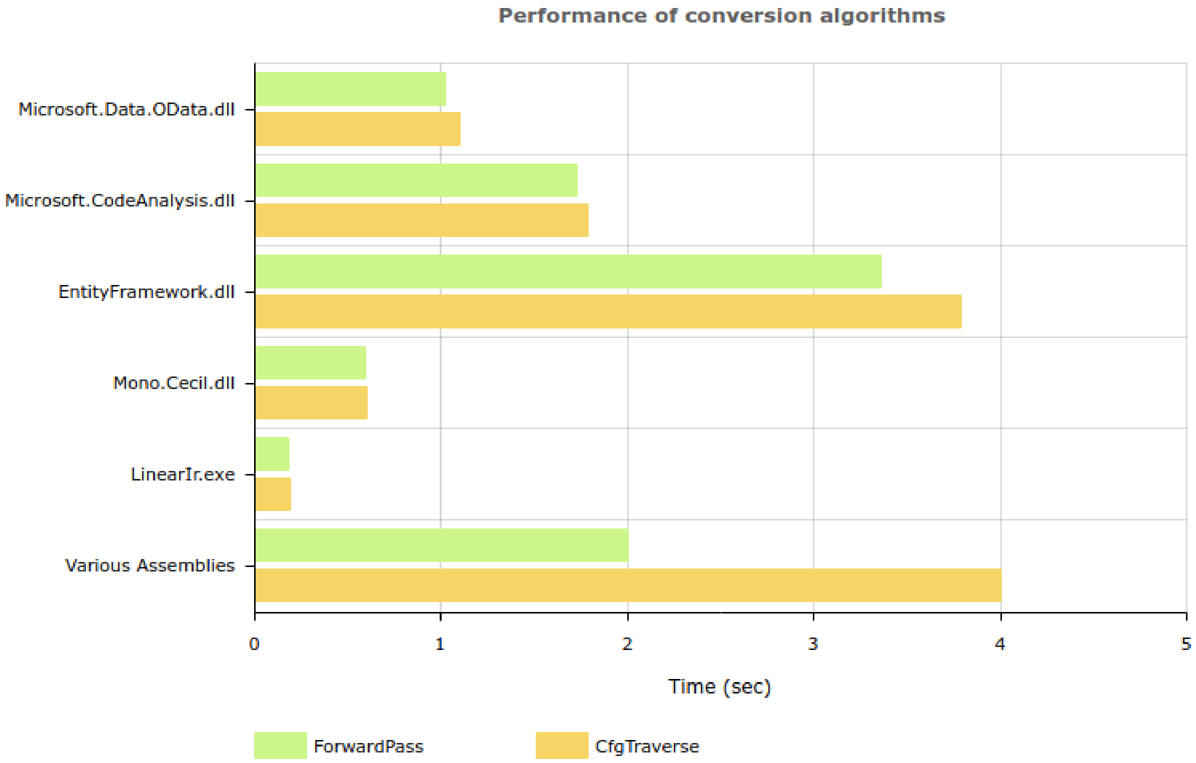


Figure 6.3: Performance of the two conversion implementations on various libraries and executables. The last category covers about 200.000 methods from various modules. We observe that *ForwardPass* performs better in all cases. As the size of the module increases, the forward pass implementation becomes noticeably better. For example, in the last case the conversion is accelerated 2 times.

Our benchmarks will be performed on *dynamically linked libraries* (dll). We will obtain some large *dlls* from arbitrary tools and frameworks. This will help us discover any edge cases and bugs too, due to the large variety of instruction sequences that this software contains.

Chart 6.3 illustrates the results. Our assumptions regarding the two algorithms were correct. The second approach we examined is more simple and efficient.

6.3 Applications

There are various techniques for performing static analysis on programs. One of them is through inference on a set of *datalog* relations. A framework for static analysis that uses this approach is *Doop*. Doop's primary defining feature is its use of Datalog for its analyses and its explicit representation of relations as tables instead of Binary Decision Diagrams (BDDs) which havebeen considered necessary for scalable points-to analysis [5].

The first step of such an analysis, is to get a datalog representation of instructions and their properties. Without going into more detail, an example of such preprocessing is the following datalog predicates [6]:

```
add_instruction:first_operandInsn= Left -> add_instruction(Insn), operand(Left).
add_instruction:second_operandInsn= Right -> add_instruction(Insn), operand(Right).
```

The above, describes that the add instruction has 2 operands. Many similar predicates can be defined. Following the pre-processing step a simple pointer analysis can be expressed entirely in Datalog as a transitive closure computation:

```
VarPointsTo(?heap, ?var) <- AssignHeapAllocation(?heap, ?var).
VarPointsTo(?heap, ?to) <- Assign(?to, ?from), VarPointsTo(?heap, ?from).
```

We observe that these kind of analysis, rely on operations to have a *source* and *destination* argument. If we were to perform such analysis on the stack-based CIL IR we would find difficulties. What is the source/destination? At this point we understand why a register-based representation is sometimes useful. the *variables* of any analysis are equivalent to *registers*. Possible future work could include the generation of predicates for the CIL instructions and an arbitrary analysis for CIL methods.

Aside from the conversion to Linear IR, our work contained an implementation of a CFG for CIL methods. This CFG can be used for many purposes such as reachability analysis, verification algorithms and code optimization.

ABBREVIATIONS - ACRONYMS

VES	Virtual Execution System
CLI	Common Language Infrastructure
CIL	Common Intermediate Language
CTS	Common Type System
PE	Portable Executable
IL	Intermediate Language
IR	Intermediate Representation
ECMA	European Computer Manufacturers Association
CFG	Control Flow Graph
DFS	Depth First Search
BFS	Breadth First Search
BB	Basic Block

APPENDIX A. INSTRUCTION SET

Table A.1: Evaluation stack popping behaviours

Pop0	no inputs
Pop1	one value type specified by data flow
Pop1+Pop1	two input values, types specified by data flow
Popi	one machine-sized integer
Popi+Pop1	Top of stack is described by data flow, next item is a native pointer
Popi+Popi	Top two items on stack are integers (size can vary by instruction)
Popi+Popi+Popi	Top three items on stack are machine-sized integers
Popi8+Pop8	Top of stack is an 8-byte integer, next is a native pointer
Popi+PopR4	Top of stack is a 4-byte floating point number, next is a native pointer
Popi+PopR8	Top of stack is an 8-byte floating point number, next is a native pointer
PopRef	Top of stack is an object reference
PopRef+Popi	Top of stack is an integer (size can vary by instruction), next is an object reference
PopRef+Popi+Popi	Top of stack has two integers (size can vary by instruction), next is an object reference
PopRef+Popi+Popi8	Top of stack is an 8-byte integer, then a native-sized integer, then an object reference
PopRef+Popi+PopR4	Top of stack is an 4-byte floating point number, then a native-sized integer, then an object reference
PopRef+Popi+PopR8	Top of stack is an 8-byte floating point number, then a native-sized integer, then an object reference
VarPop	variable number of items used. For example, *call* insruction.

Table A.3: Instruction Set

Instruction	**ControlFlow**	**StakBehaviourPop**	**StackBehaviourPush**
nop	Next	Pop0	Push0
break	Break	Pop0	Push0
ldarg.0	Next	Pop0	Push1
ldarg.1	Next	Pop0	Push1
ldarg.2	Next	Pop0	Push1
ldarg.3	Next	Pop0	Push1
ldloc.0	Next	Pop0	Push1
ldloc.1	Next	Pop0	Push1
ldloc.2	Next	Pop0	Push1
ldloc.3	Next	Pop0	Push1
stloc.0	Next	Pop1	Push0
stloc.1	Next	Pop1	Push0

stloc.2	Next	Pop1	Push0
stloc.3	Next	Pop1	Push0
ldarg.s	Next	Pop0	Push1
ldarga.s	Next	Pop0	Pushi
starg.s	Next	Pop1	Push0
ldloc.s	Next	Pop0	Push1
ldloca.s	Next	Pop0	Pushi
stloc.s	Next	Pop1	Push0
ldnull	Next	Pop0	Pushref
ldc.i4.m1	Next	Pop0	Pushi
ldc.i4.0	Next	Pop0	Pushi
ldc.i4.1	Next	Pop0	Pushi
ldc.i4.2	Next	Pop0	Pushi
ldc.i4.3	Next	Pop0	Pushi
ldc.i4.4	Next	Pop0	Pushi
ldc.i4.5	Next	Pop0	Pushi
ldc.i4.6	Next	Pop0	Pushi
ldc.i4.7	Next	Pop0	Pushi
ldc.i4.8	Next	Pop0	Pushi
ldc.i4.s	Next	Pop0	Pushi
ldc.i4	Next	Pop0	Pushi
ldc.i8	Next	Pop0	Pushi8
ldc.r4	Next	Pop0	Pushr4
ldc.r8	Next	Pop0	Pushr8
dup	Next	Pop1	Push1_push1
pop	Next	Pop1	Push0
jmp	Call	Pop0	Push0
call	Call	Varpop	Varpush
calli	Call	Varpop	Varpush
ret	Return	Varpop	Push0
br.s	Branch	Pop0	Push0
brfalse.s	Cond_Branch	Popi	Push0
brtrue.s	Cond_Branch	Popi	Push0
beq.s	Cond_Branch	Pop1_pop1	Push0
bge.s	Cond_Branch	Pop1_pop1	Push0
bgt.s	Cond_Branch	Pop1_pop1	Push0
ble.s	Cond_Branch	Pop1_pop1	Push0
blt.s	Cond_Branch	Pop1_pop1	Push0
bne.un.s	Cond_Branch	Pop1_pop1	Push0
bge.un.s	Cond_Branch	Pop1_pop1	Push0
bgt.un.s	Cond_Branch	Pop1_pop1	Push0
ble.un.s	Cond_Branch	Pop1_pop1	Push0
blt.un.s	Cond_Branch	Pop1_pop1	Push0
br	Branch	Pop0	Push0
brfalse	Cond_Branch	Popi	Push0
brtrue	Cond_Branch	Popi	Push0
beq	Cond_Branch	Pop1_pop1	Push0
bge	Cond_Branch	Pop1_pop1	Push0

bgt	Cond_Branch	Pop1_pop1	Push0
ble	Cond_Branch	Pop1_pop1	Push0
blt	Cond_Branch	Pop1_pop1	Push0
bne.un	Cond_Branch	Pop1_pop1	Push0
bge.un	Cond_Branch	Pop1_pop1	Push0
bgt.un	Cond_Branch	Pop1_pop1	Push0
ble.un	Cond_Branch	Pop1_pop1	Push0
blt.un	Cond_Branch	Pop1_pop1	Push0
switch	Cond_Branch	Popi	Push0
ldind.i1	Next	Popi	Pushi
ldind.u1	Next	Popi	Pushi
ldind.i2	Next	Popi	Pushi
ldind.u2	Next	Popi	Pushi
ldind.i4	Next	Popi	Pushi
ldind.u4	Next	Popi	Pushi
ldind.i8	Next	Popi	Pushi8
ldind.i	Next	Popi	Pushi
ldind.r4	Next	Popi	Pushr4
ldind.r8	Next	Popi	Pushr8
ldind.ref	Next	Popi	Pushref
stind.ref	Next	Popi_popi	Push0
stind.i1	Next	Popi_popi	Push0
stind.i2	Next	Popi_popi	Push0
stind.i4	Next	Popi_popi	Push0
stind.i8	Next	Popi_popi8	Push0
stind.r4	Next	Popi_popr4	Push0
stind.r8	Next	Popi_popr8	Push0
add	Next	Pop1_pop1	Push1
sub	Next	Pop1_pop1	Push1
mul	Next	Pop1_pop1	Push1
div	Next	Pop1_pop1	Push1
div.un	Next	Pop1_pop1	Push1
rem	Next	Pop1_pop1	Push1
rem.un	Next	Pop1_pop1	Push1
and	Next	Pop1_pop1	Push1
or	Next	Pop1_pop1	Push1
xor	Next	Pop1_pop1	Push1
shl	Next	Pop1_pop1	Push1
shr	Next	Pop1_pop1	Push1
shr.un	Next	Pop1_pop1	Push1
neg	Next	Pop1	Push1
not	Next	Pop1	Push1
conv.i1	Next	Pop1	Pushi
conv.i2	Next	Pop1	Pushi
conv.i4	Next	Pop1	Pushi
conv.i8	Next	Pop1	Pushi8
conv.r4	Next	Pop1	Pushr4
conv.r8	Next	Pop1	Pushr8

conv.u4	Next	Pop1	Pushi
conv.u8	Next	Pop1	Pushi8
callvirt	Call	Varpop	Varpush
cpobj	Next	Popi_popi	Push0
ldobj	Next	Popi	Push1
ldstr	Next	Pop0	Pushref
newobj	Call	Varpop	Pushref
castclass	Next	Popref	Pushref
isinst	Next	Popref	Pushi
conv.r.un	Next	Pop1	Pushr8
unbox	Next	Popref	Pushi
throw	Throw	Popref	Push0
ldfld	Next	Popref	Push1
ldflda	Next	Popref	Pushi
stfld	Next	Popref_pop1	Push0
ldsfld	Next	Pop0	Push1
ldsflda	Next	Pop0	Pushi
stsfld	Next	Pop1	Push0
stobj	Next	Popi_pop1	Push0
conv.ovf.i1.un	Next	Pop1	Pushi
conv.ovf.i2.un	Next	Pop1	Pushi
conv.ovf.i4.un	Next	Pop1	Pushi
conv.ovf.i8.un	Next	Pop1	Pushi8
conv.ovf.u1.un	Next	Pop1	Pushi
conv.ovf.u2.un	Next	Pop1	Pushi
conv.ovf.u4.un	Next	Pop1	Pushi
conv.ovf.u8.un	Next	Pop1	Pushi8
conv.ovf.i.un	Next	Pop1	Pushi
conv.ovf.u.un	Next	Pop1	Pushi
box	Next	Pop1	Pushref
newarr	Next	Popi	Pushref
ldlen	Next	Popref	Pushi
ldelema	Next	Popref_popi	Pushi
ldelem.i1	Next	Popref_popi	Pushi
ldelem.u1	Next	Popref_popi	Pushi
ldelem.i2	Next	Popref_popi	Pushi
ldelem.u2	Next	Popref_popi	Pushi
ldelem.i4	Next	Popref_popi	Pushi
ldelem.u4	Next	Popref_popi	Pushi
ldelem.i8	Next	Popref_popi	Pushi8
ldelem.i	Next	Popref_popi	Pushi
ldelem.r4	Next	Popref_popi	Pushr4
ldelem.r8	Next	Popref_popi	Pushr8
ldelem.ref	Next	Popref_popi	Pushref
stelem.i	Next	Popref_popi_popi	Push0
stelem.i1	Next	Popref_popi_popi	Push0
stelem.i2	Next	Popref_popi_popi	Push0
stelem.i4	Next	Popref_popi_popi	Push0

stelem.i8	Next	Popref_popi_popi8	Push0
stelem.r4	Next	Popref_popi_popr4	Push0
stelem.r8	Next	Popref_popi_popr8	Push0
stelem.ref	Next	Popref_popi_popref	Push0
ldelem.any	Next	Popref_popi	Push1
stelem.any	Next	Popref_popi_popref	Push0
unbox.any	Next	Popref	Push1
conv.ovf.i1	Next	Pop1	Pushi
conv.ovf.u1	Next	Pop1	Pushi
conv.ovf.i2	Next	Pop1	Pushi
conv.ovf.u2	Next	Pop1	Pushi
conv.ovf.i4	Next	Pop1	Pushi
conv.ovf.u4	Next	Pop1	Pushi
conv.ovf.i8	Next	Pop1	Pushi8
conv.ovf.u8	Next	Pop1	Pushi8
refanyval	Next	Pop1	Pushi
ckfinite	Next	Pop1	Pushr8
mkrefany	Next	Popi	Push1
ldtoken	Next	Pop0	Pushi
conv.u2	Next	Pop1	Pushi
conv.u1	Next	Pop1	Pushi
conv.i	Next	Pop1	Pushi
conv.ovf.i	Next	Pop1	Pushi
conv.ovf.u	Next	Pop1	Pushi
add.ovf	Next	Pop1_pop1	Push1
add.ovf.un	Next	Pop1_pop1	Push1
mul.ovf	Next	Pop1_pop1	Push1
mul.ovf.un	Next	Pop1_pop1	Push1
sub.ovf	Next	Pop1_pop1	Push1
sub.ovf.un	Next	Pop1_pop1	Push1
endfinally	Return	Pop0	Push0
leave	Branch	PopAll	Push0
leave.s	Branch	PopAll	Push0
stind.i	Next	Popi_popi	Push0
conv.u	Next	Pop1	Pushi
arglist	Next	Pop0	Pushi
ceq	Next	Pop1_pop1	Pushi
cgt	Next	Pop1_pop1	Pushi
cgt.un	Next	Pop1_pop1	Pushi
clt	Next	Pop1_pop1	Pushi
clt.un	Next	Pop1_pop1	Pushi
ldftn	Next	Pop0	Pushi
ldvirtftn	Next	Popref	Pushi
ldarg	Next	Pop0	Push1
ldarga	Next	Pop0	Pushi
starg	Next	Pop1	Push0
ldloc	Next	Pop0	Push1
ldloca	Next	Pop0	Pushi

stloc	Next	Pop1	Push0
localloc	Next	Popi	Pushi
endfilter	Return	Popi	Push0
unaligned.	Meta	Pop0	Push0
volatile.	Meta	Pop0	Push0
tail.	Meta	Pop0	Push0
initobj	Next	Popi	Push0
constrained.	Next	Pop0	Push0
cpblk	Next	Popi_popi_popi	Push0
initblk	Next	Popi_popi_popi	Push0
no.	Next	Pop0	Push0
rethrow	Throw	Pop0	Push0
sizeof	Next	Pop0	Pushi
refanytype	Next	Pop1	Pushi
readonly.	Next	Pop0	Push0

Table A.2: Evaluation stack pushing behaviours

Push0	no output value
Push1	one output value, type defined by data flow.
Push1+Push1	two output values, type defined by data flow
Pushi	push one native integer or pointer
Pushi8	push one 8-byte integer
PushR4	push one 4-byte floating point number
PushR8	push one 8-byte floating point number
PushRef	push one object reference
VarPush	variable number of items pushed, for example, *ret* instruction.

REFERENCES

[1] *Microsoft .NET Documentation*.

[2] *Mono Documentation*.

[3] *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 2006.

[4] *Standard ECMA-335 Common Language Infrastructure (CLI)*, 2012.

[5] Nikolaos Filippakis. Static dependence analysis for java.

[6] Eirini I. Psallida. Relational representation of the llvm intermediate language.

[7] Yannis Smaragdakis. Compiler optimization lectures.