David Srbecký
Jesus College
ds417

Part II of the Computer Science Project Proposal

.NET Decompiler

October 14, 2007

Project Originator: *David Srbecký*

Resources Required: See attached Project Resource Form

Project Supervisor: *Alan Mycroft*

Signature:

Director of Studies: *Jean Bacon* and *David Ingram*

Signature:

Overseers: *Anuj Dawar* and *Andrew Moore*

Signatures:

Introduction and Description of the Work

The *.NET Framework* is a general-purpose software development platform which is very similar to *Java*. It includes extensive class library and, similarly to Java, is based on the virtual machine model. The executable code for a *.NET* program is stored in a file called *assembly* which consists of class metadata and a stack-based bytecode called Common Intermediate Language (*CIL* or *IL*).

In general, any programming language can be compiled to *.NET* and there are dozens of compilers that compile into *CIL*. The most common language used for *.NET* development is *C#*.

The goal of this project is to decompile *.NET* assemblies back into equivalent *C#* source code. Compared to decompilation of conventional assembly code, this task is hugely simplified by the presence of metadata in the *.NET* assemblies. The metadata contains complete information about classes, methods and fields. The method bodies consist of stack-based *IL* code which needs to be decompiled into higher-level *C#* statements. Data-flow analysis will need to be employed to transform the stack-based data model into one that uses temporary local variables and composition of expressions. Control-flow analysis will be used to recreate high level control structures like `for` loops and conditional branching.

Resources Required

- **My own machine**
  (1.6 GHz CPU, 1.5 GB of RAM, 50 GB & 75 GB Disks, Windows XP SP2 OS)
  Used for development
- **Student-Run Computing Facility (SRCF)**
  Used for running the *SVN* server
- **Public Workstation Facility (PWF)**
  Used for storage of back-ups

Starting Point

I plan to implement the project in *C#*. I have been using this language for over five years now and so I do not have to spend any time learning a new language. It also means that I will not be having any problems neither with the syntax of the language nor with any peculiar error messages produced by the compiler or by the runtime.

I have written an integrated *.NET* debugger for the *SharpDevelop* IDE. During that I have obtained some basic knowledge about metadata and lower-level functionality in *.NET*. I can read *.NET* bytecode and, with the help of reference manual, I can write short programs in it.

The metadata and bytecode needs to be read form the assembly files. I plan to use the *Cecil* library for it. I am not familiar with this library, but I do not expect to have any difficulties with it.

Substance and Structure of the Project

The project consists of the following major work items:

1. **Preliminary research**
   I will have to research the following topics:

   - *Cecil* library - *Cecil* is the library which I will use for reading of the metadata. It will need to get familiar with its public API. Because it is open-source, it might be valuable to get some basic understanding of its source code as well.
   - *CIL* bytecode - The runtime of the *.NET Framework* is described in ECMA-335 Standard: *"CLI Specification – Virtual Machine"* (556 pages). I will need to get familiar with this document since I will be using it as the main reference. I will be especially interested in *Partition III – CIL Instruction Set*.
   - Decompilation theory - I will need to get familiar with the theory behind decompilation of programs. Cristina Cifuentes' PhD thesis *"Reverse Compilation Techniques"* might prove as especially useful starting point.

   The research of these topics should not be too extensive. I only indeed to get sufficient background knowledge in these areas and then return to the finner details when I needed them.

2. **Create a skeleton of the code**
   It will be necessary to read the assembly metadata and create a *C#* source code that has the same classes, fields and methods. The method signatures have to match the ones in the assembly. At this point the method bodies can be left empty.

3. **Read and disassemble *.NET* bytecode**
   The next step is to read the bytecode for each method, disassemble it and output it as comments (for example, `// IL_01: ldstr "Hello world"`). This will help me learn how to use the *Cecil* library to read the bytecode and how to process it. I also expect that this output will be extremely helpful for debugging purposes later on.

4. **Start creating r-value expressions**
   Ignoring the stack of the virtual machine, some bytecodes can be straightforwardly converted into expressions. For example:

```
ldstr "Hello world"      - string "Hello world"
ldnull                   - 'null' reference
ldc.i4.0                 - 4 byte integer of value 0
ldc.i4 123               - 4 byte integer of value 123
ldarg.0                  - the first method argument
ldloc.0                  - the first local variable in the method
```

   The goal of this stage is to create *C#* expressions for several of the most important bytecodes.

   Function calls and arithmetic operations are also expressions, but at this stage I do not know their inputs and so I will have to use dummy values as their inputs.

5. **Conditional and unconditional branching**
   There are several bytecodes that investigate one or two values on the top of stack and then, if a given condition is met, branch to different location. (`br`, `brfalse`, `brtrue`, `beq`, `bge`, `bgt`, etc...)

   The goal of this stage is to use *C#* labels and `goto` statements to recreate this flow of control. (eg translate `brfalse IL_02` to `if (input == false) goto Label_02;`)

   As in the previous stage the inputs (ie the values at the top of stack) are still not know.

6. **Simple data-flow analysis**
   This is where it begins to be difficult. Consider the code:

```
// Load "Hello, world!" on top of the stack
IL_01: ldstr "Hello, world!"
// Print the top of the stack to the console
IL_02: call void [mscorlib]System.Console::WriteLine(string)
```

   Both of these are already decompiled as expressions, however the call has a dummy value as its argument. The goal of this stage is to perform as simple data-flow analysis as possible. The text "Hello, world!" must find its way to the method call. At this point it will probably be through one or even two temporary variables. For example:

```
String il_01_expression = "Hello, world!";
String il_02_argument_1 = il_01_expression;
System.Console.WriteLine(il_02_argument_1);
```

The most difficult part will be handling of control flow. Different values can be on stack depending on which branch of code was executed. At this stage it will be necessary to create and analyse control flow graph. As a result of this stage, many temporary variables might be introduced to the code.

7. **Round-trip quick-sort algorithm**
   At this point very simple applications should probably successfully decompile and compile again (round-trip).

   The goal of this stage is to fix bugs and to add features so that simple algorithm like quick-sort can be successfully round-tripped without need to manually change the produced *C#* source code. At this point there is no restriction on the aesthetics of the source code. The only requirement is that it does compile.

   There are many features of *.NET* that I do not plan to support at this point. For example, boxing & unboxing, casting, generics and exception handling. In general, all non-essential features are excluded.

8. **Further data-flow analysis**
   Employ more advanced data-flow analysis to simplify the generated *C#* code. Many temporary variables can be probably removed, relocated or renamed according to their use.

   [This task has variable scope and if the project starts falling behind schedule, simpler algorithms can be employed and vice versa.]

9. **Control-flow analysis**
   The goal of this stage is to use control-flow analysis to regenerate high-level structures like `if` statements and `for` loops. It will not be possible to eliminated all `goto` statements, but they should be avoided whenever possible.

   [This task has variable scope and if the project starts falling behind schedule, simpler algorithms can be employed and vice versa.]

10. **Assembly resources**
    .NET assemblies can have files embed in them. These files can then be accessed at runtime and thus the programs might require them.

    The goal is to extract the resources so that they can be included during the recompilation process.

    [Optional. This is an optional goal which will be done only if the project development goes much better then originally anticipated.]

11. **Advanced features**
    Add commonly used features which where ignored so far - for example, boxing & unboxing, casting, generics and exception handling.

    [Optional. This is an optional goal which will be done only if the project development goes much better then originally anticipated.]

12. **Round-trip Mono**
    The ultimate goal of this project is to be able to round-trip any *.NET* assembly. This means that for any given assembly the Decompiler should produce *C#* source code which is valid (does compile again without error). Even more importantly, the program produced by the compilation of the source code should be semantically same as the original one. Since the bytecode will in general differ, this condition is difficult to verify. One way to check that the Decompiler preserves the meaning of programs is to simply try it.

    Mono is open-source reimplantation of the *.NET Framework*. The major part of it are the *.NET* class libraries which can be used for testing of the Decompiler. The project is open-source and so if any decompilation problems occur, it is possible to investigate the source code of these libraries. Furthermore, the libraries come with extensive unit testing suite so it is possible to verify that the round-tripped libraries are not broken.

    The goal of this final stage is to successfully round-trip all *Mono* libraries and pass the unit tests. This would probably involve enormous amount of bugfixing, investigation and handling of corner cases. All remaining *.NET* features would have to be implemented.

    [Optional. This last stage is huge and impossible to be finished within the time frame of Part II project. If all goes well, I expect that it will take at least one more year for the project to mature to this point.]

13. **Write the dissertation**
    The last and most important piece of work is to write the dissertation. Being a non-native English speaker, I expect this to take considerable amount of time. I plan to spend the last seven weeks of project time on it. This includes the end of Lent Term and the whole Easter vacation. I plan to have the dissertation finished by the start of Easter term.

Success Criteria

The Decompiler should successfully round-trip a quick-sort algorithm (or any algorithm of comparable complexity). That is, when an assembly containing the algorithm is decompiled, the produced *C#* source code should be both syntactically and semantically correct. The bytecode produced by compilation of the generated source code is not expected to be identical to the original one, but it is expected to be equivalent. That is, the binary may be different but it still needs to be a correct implementation of the algorithm.

To achieve this the Decompiler will need to have the following features:

- Handle integers and integer arithmetic
- Create and be able to use integer arrays
- Branching must be successfully decompiled
- Several methods can be defined
- Methods can have arguments and return values
- Methods can be called recursively
- Integer command line arguments can be read and parsed
- Text can be outputted to the standard console output

See the following page for a *C#* implementation of a quick-sort algorithm which will be used to demonstrate successful implementation of these features.

I plan to achieve the success criteria by the progress report dead-line and then spend the rest of the time available by increasing the quality of the generated source code (ie "Further data-flow analysis" and "Control-flow analysis").

Quick-sort algorithm

```
1  static class QuickSortProgram
2  {
3      public static void Main(string[] args)
4      {
5          int[] intArray = new int[args.Length];
6          for (int i = 0; i < intArray.Length; i++) {
7              intArray[i] = int.Parse(args[i]);
8          }
9          QuickSort(intArray, 0, intArray.Length - 1);
10         for (int i = 0; i < intArray.Length; i++) {
11             System.Console.Write(intArray[i].ToString() + " ");
12         }
13     }
14
15     /// For description of this algorithm see:
16     /// http://en.wikipedia.org/wiki/Quick_sort
17     public static void QuickSort(int[] array, int left, int right)
18     {
19         if (right > left) {
20             int pivotIndex = (left + right) / 2;
21             int pivotNew = Partition(array, left, right, pivotIndex);
22             QuickSort(array, left, pivotNew - 1);
23             QuickSort(array, pivotNew + 1, right);
24         }
25     }
26
27     static int Partition(int[] array, int left, int right,
28                          int pivotIndex)
29     {
30         int pivotValue = array[pivotIndex];
31         Swap(array, pivotIndex, right);
32         int storeIndex = left;
33         for(int i = left; i < right; i++) {
34             if (array[i] <= pivotValue) {
35                 Swap(array, storeIndex, i);
36                 storeIndex = storeIndex + 1;
37             }
38         }
39         Swap(array, right, storeIndex);
40         return storeIndex;
41     }
42
43     static void Swap(int[] array, int index1, int index2)
44     {
45         int tmp = array[index1];
46         array[index1] = array[index2];
47         array[index2] = tmp;
48     }
49 }
```

Timetable and Milestones

The work shall start on the Monday 22.10.2007 and is expected to take 20 weeks in total.

22 Oct - 28 Oct	*(week 1)*	**Preliminary research**
29 Oct - 4 Nov	*(week 2)*	**Create a skeleton of the code**
5 Nov - 11 Nov	*(week 3)*	**Read and disassemble *.NET* bytecode**
12 Nov - 18 Nov	*(week 4)*	**Start creating r-value expressions**
19 Nov - 25 Nov	*(week 5)*	**Conditional and unconditional branching**
26 Nov - 9 Dec	*(weeks 6 and 7)*	**Simple data-flow analysis**
10 Dec - 20 Jan		Christmas vacation
21 Jan - 27 Jan	*(week 8)*	**Round-trip quick-sort algorithm**
26 Jan - 27 Jan		**Write the Progress Report**
28 Jan - 10 Feb	*(weeks 9 and 10)*	**Further data-flow analysis**
11 Feb - 2 Mar	*(weeks 11 to 13)*	**Control-flow analysis**
3 Mar - 20 Apr	*(weeks 14 to 20)*	**Write the dissertation** (over Easter vacation)
21 Apr onwards		Easter term – Preparation for exams

Unscheduled tasks: **Assembly resources**; **Advanced features**; **Round-trip Mono**