David Srbecký
Jesus College
ds417

Progress Report

.NET Decompiler

January 30, 2008

Project Originator: *David Srbecký*

Project Supervisor: *Alan Mycroft*

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

Overseers: *Anuj Dawar* and *Andrew Moore*

Work completed so far

Disassemble *.NET* bytecode

The *.NET* assembly is read using the *Cecil* library and the class structure is created. Method bodies contain the disassembly of the IL bytecode. The debugging comment on the right indicates the stack behavior of the given instruction. This, of course, is not valid *C#* code yet.

```
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        IL_00: nop                  # Pop0->Push0
        IL_01: ldarg args           # Pop0->Push1
        IL_02: ldlen                # Popref->Pushi
        IL_03: conv.i4              # Pop1->Pushi
        IL_04: newarr System.Int32# Popi->Pushref
        IL_09: stloc V_0            # Pop1->Push0
        IL_0A: ldc.i4 0             # Pop0->Pushi
        IL_0B: stloc V_1            # Pop1->Push0
        IL_0C: br IL_1F             # Pop0->Push0 Flow=Branch
        IL_0E: nop                  # Pop0->Push0
        IL_0F: ldloc V_0            # Pop0->Push1
        IL_10: ldloc V_1            # Pop0->Push1
        IL_11: ldarg args           # Pop0->Push1
        IL_12: ldloc V_1            # Pop0->Push1
        IL_13: ldelem.ref           # Popref_popi->Pushref
        IL_14: call Parse()         # Varpop->Varpush Flow=Call
        IL_19: stelem.i4            # Popref_popi_popi->Push0
        IL_1A: nop                  # Pop0->Push0
        IL_1B: ldloc V_1            # Pop0->Push1
        IL_1C: ldc.i4 1             # Pop0->Pushi
        IL_1D: add.ovf              # Pop1_pop1->Push1
        IL_1E: stloc V_1            # Pop1->Push0
        IL_1F: ldloc V_1            # Pop0->Push1
        IL_20: ldloc V_0            # Pop0->Push1
        IL_21: ldlen                # Popref->Pushi
        IL_22: conv.i4              # Pop1->Pushi
        IL_23: clt                  # Pop1_pop1->Pushi
        IL_25: stloc V_2            # Pop1->Push0
```

Start creating expressions

The bytecodes are converted to *C#* expressions on individual basis. Only one bytecode is considered at a time and thus the expressions are completely independent. The resulting output is a valid *C#* code which however does not compile since the dummy arguments `arg1`, `arg2`, etc... are never defined. Conditional and unconditional branches are converted to `goto` goto statements.

```
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        IL_00: // No-op
        IL_01: System.String[] expr01 = args;
        IL_02: int expr02 = arg1.Length;
        IL_03: int expr03 = (Int32)arg1;
        IL_04: object expr04 = new int[arg1];
        IL_09: V_0 = arg1;
        IL_0A: short expr0A = 0;
        IL_0B: V_1 = arg1;
        IL_0C: goto IL_1F;
        IL_0E: // No-op
        IL_0F: System.Int32[] expr0F = V_0;
        IL_10: int expr10 = V_1;
        IL_11: System.String[] expr11 = args;
        IL_12: int expr12 = V_1;
        IL_13: object expr13 = arg1[arg2];
        IL_14: int expr14 = System.Int32.Parse(arg0);
        IL_19: arg1[arg2] = arg3;
        IL_1A: // No-op
        IL_1B: int expr1B = V_1;
        IL_1C: short expr1C = 1;
        IL_1D: int expr1D = arg1 + arg2;
        IL_1E: V_1 = arg1;
        IL_1F: int expr1F = V_1;
        IL_20: System.Int32[] expr20 = V_0;
        IL_21: int expr21 = arg1.Length;
        IL_22: int expr22 = (Int32)arg1;
        IL_23: bool expr23 = arg1 < arg2;
        IL_25: V_2 = arg1;
```

Data-flow analysis

The execution of the bytecode is simulated and the state of the stack is recorded for each position. We are interested in the number of elements on the stack as well as which instruction has pushed the individual elements on the stack. This information can then be used to eliminate the dummy `arg1` arguments. Result of each instruction is stored in new temporary variable. When an instruction pops a stack value we look-up which instruction has allocated the value and use the temporary variable of the allocating instruction. This code compiles and works correctly.

```
        int expr0A = 0;
        // Stack: {expr0A}
        V_1 = expr0A;
        // Stack: {}
        goto IL_1F;
        // Stack: {}
        IL_0E: // No-op
        // Stack: {}
        System.Int32[] expr0F = V_0;
        // Stack: {expr0F}
        int expr10 = V_1;
        // Stack: {expr0F, expr10}
        System.String[] expr11 = args;
        // Stack: {expr0F, expr10, expr11}
        int expr12 = V_1;
        // Stack: {expr0F, expr10, expr11, expr12}
        string expr13 = expr11[expr12];
        // Stack: {expr0F, expr10, expr13}
        int expr14 = System.Int32.Parse(expr13);
        // Stack: {expr0F, expr10, expr14}
        expr0F[expr10] = expr14;
        // Stack: {}
        // No-op
        // Stack: {}
        int expr1B = V_1;
        // Stack: {expr1B}
        int expr1C = 1;
        // Stack: {expr1B, expr1C}
        int expr1D = expr1B + expr1C;
        // Stack: {expr1D}
        V_1 = expr1D;
        // Stack: {}
```

In-lineing expressions

Many of the temporary variables can be in-lined into the expressions in which they are used. This is in general non-trivial optimization, however it is simpler in this case since the temporary variables generated to store the stack values are guaranteed to be single static assignment variables (the variable is assigned only once during the push instruction and is used only once during the pop instruction). Having said that, we still need to check that doing the optimization is safe with regards to expression evaluation order and with regrads to branching.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        System.Int32[] V_0 = new int[((Int32)args.Length)];
        int V_1 = 0;
        goto IL_1C;
        IL_0D: V_0[V_1] = System.Int32.Parse(args[V_1]);
        V_1 = (V_1 + 1);
        IL_1C: if (V_1 < ((Int32)V_0.Length)) goto IL_0D;
        QuickSortProgram.QuickSort(V_0, 0, (((Int32)V_0.Length) - 1));
        int V_2 = 0;
        goto IL_51;
        IL_32: System.Console.Write(System.String.Concat((V_0[V_2]).ToString(),
        V_2 = (V_2 + 1);
        IL_51: if (V_2 < ((Int32)V_0.Length)) goto IL_32;
    }
    public static void QuickSort(System.Int32[] array, int left, int right)
    {
        if (right <= left) goto IL_28;
        int V_0 = ((left + right) / 2);
        int V_1 = QuickSortProgram.Partition(array, left, right, V_0);
        QuickSortProgram.QuickSort(array, left, (V_1 - 1));
        QuickSortProgram.QuickSort(array, (V_1 + 1), right);
        IL_28: return;
    }
    private static int Partition(System.Int32[] array, int left, int right, int
    {
        int V_0 = array[pivotIndex];
        QuickSortProgram.Swap(array, pivotIndex, right);
        int V_1 = left;
```

Finding basic blocks

The first step of reconstructing any high-level structures is the decomposition of the program into basic blocks. This is an easy algorithm to implement.

I chose to use the following constraint for the output: “Each basic block starts with a label and is exited by an explicit `goto` statement.” Therefore except for the method entry, the order of the blocks is completely irrelevant. Any swapping of the basic blocks is not going change the semantics of the program in any way.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        BasicBlock_1:
        System.Int32[] V_0 = new int[((int)args.Length)];
        int i = 0;
        goto BasicBlock_3;
        BasicBlock_2:
        V_0[i] = System.Int32.Parse(args[i]);
        i = (i + 1);
        goto BasicBlock_3;
        BasicBlock_3:
        if (i < ((int)V_0.Length)) goto BasicBlock_2;
        goto BasicBlock_4;
        BasicBlock_4:
        QuickSortProgram.QuickSort(V_0, 0, (((int)V_0.Length) - 1));
        int j = 0;
        goto BasicBlock_6;
        BasicBlock_5:
        System.Console.Write(System.String.Concat((V_0[j]).ToString(), " "));
        j = (j + 1);
        goto BasicBlock_6;
        BasicBlock_6:
        if (j < ((int)V_0.Length)) goto BasicBlock_5;
        goto BasicBlock_7;
        BasicBlock_7:
        return;
    }
```

Finding loops

The algorithm for finding loops is inspired by T1-T2 transformations. T1-T2 transformations are used to determine whether a graph is reducible or not. The core idea is that if a block of code has only one predecessor then the block of code can be merged with its predecessor to form a directed acyclic graph. Using this, loops will reduce to single self-referencing nodes. This also works for nested loops.

Note that merely adding a loop does not change the program in any way – the loop is completely redundant as far as control flow goes. The basic blocks still explicitly transfer control using `goto` statements, so the control flow never reaches the loop.

This is desirable property. It ensures that the program will run correctly. The order of basic blocks and their nesting within loops does not have any effect on program correctness.

The only advantage of the loop is readability and that some `goto` statements can be replaced by `break` and `continue` statements if they have the same semantics in the given context.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        BasicBlock_1:
        System.Int32[] V_0 = new int[((int)args.Length)];
        int i = 0;
        goto Loop_8;
        Loop_8:
        for (;;) {
            BasicBlock_3:
            if (i < ((int)V_0.Length)) goto BasicBlock_2;
            break;
            BasicBlock_2:
            V_0[i] = System.Int32.Parse(args[i]);
            i = (i + 1);
            continue;
        }
        BasicBlock_4:
        QuickSortProgram.QuickSort(V_0, 0, (((int)V_0.Length) - 1));
        int j = 0;
        goto Loop_11;
        Loop_11:
        for (;;) {
```

Finding conditionals

The current algorithm for finding conditionals works as follows: First find a node that has two successors. Get all nodes accessible *only* from the 'true' branch – these form the 'true' body of the conditional. Similarly, all nodes accessible *only* from the 'false' branch form the 'false' body. The rest of the nodes is not part of the conditional.

Similarly as for the loops, adding a conditional does not have any effect on program correctness.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        BasicBlock_1:
        System.Int32[] V_0 = new int[((int)args.Length)];
        int i = 0;
        goto Loop_8;
        Loop_8:
        for (;;) {
            ConditionalNode_16:
            BasicBlock_3:
            if (!(i < ((int)V_0.Length))) {
                break;
                Block_14:
            }
            else {
                goto BasicBlock_2;
                Block_15:
            }
            BasicBlock_2:
            V_0[i] = System.Int32.Parse(args[i]);
            i = (i + 1);
            continue;
        }
        BasicBlock_4:
        QuickSortProgram.QuickSort(V_0, 0, (((int)V_0.Length) - 1));
        int j = 0;
        goto Loop_11;
        Loop_11:
        for (;;) {
```

Remove dead jumps

There are many `goto` statements in the form:

```
goto BasicBlock_X;
BasicBlock_X:
```

These `goto` statement can be removed. As a result of doing that, several labels will become dead; these can be removed as well.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        System.Int32[] V_0 = new int[((int)args.Length)];
        int i = 0;
        for (;;) {
            if (!(i < ((int)V_0.Length))) {
                break;
            }
            else {
            }
            V_0[i] = System.Int32.Parse(args[i]);
            i = (i + 1);
        }
        QuickSortProgram.QuickSort(V_0, 0, (((int)V_0.Length) - 1));
        int j = 0;
        for (;;) {
            if (!(j < ((int)V_0.Length))) {
                break;
            }
            else {
            }
            System.Console.Write(System.String.Concat((V_0[j]).ToString(), " "));
            j = (j + 1);
        }
    }
    public static void QuickSort(System.Int32[] array, int left, int right)
    {
        if (!(right <= left)) {
            int i = ((left + right) / 2);
```

Reduce loops

It is common for loops to be preceded by a temporary variable initialization, start by evaluating a condition and finally end by doing an increment on a variable. We can look for these patterns and if they are found move the code to the `for(;;)` part of the statement.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(System.String[] args)
    {
        System.Int32[] V_0 = new int[((int)args.Length)];

        for (int i = 0; (i < ((int)V_0.Length)); i = (i + 1)) {
            V_0[i] = System.Int32.Parse(args[i]);
        }
        QuickSortProgram.QuickSort(V_0, 0, (((int)V_0.Length) - 1));

        for (int j = 0; (j < ((int)V_0.Length)); j = (j + 1)) {
            System.Console.Write(System.String.Concat((V_0[j]).ToString(), " "))
        }
    }
    public static void QuickSort(System.Int32[] array, int left, int right)
    {
        if (!(right <= left)) {
            int i = ((left + right) / 2);
            int j = QuickSortProgram.Partition(array, left, right, i);
            QuickSortProgram.QuickSort(array, left, (j - 1));
            QuickSortProgram.QuickSort(array, (j + 1), right);
        }
        else {
        }
    }
    private static int Partition(System.Int32[] array, int left, int right, int
    {
        int i = array[pivotIndex];
        QuickSortProgram.Swap(array, pivotIndex, right);
        int j = left;
```

Clean up

Finally some minor cleanups like removing empty statements and simplifying type names.

```
using System;
abstract class QuickSortProgram
{
    public static void Main(string[] args)
    {
        int[] V_0 = new int[((int)args.Length)];
        for (int i = 0; (i < ((int)V_0.Length)); i = (i + 1)) {
            V_0[i] = Int32.Parse(args[i]);
        }
        QuickSort(V_0, 0, (((int)V_0.Length) - 1));
        for (int j = 0; (j < ((int)V_0.Length)); j = (j + 1)) {
            Console.Write((V_0[j]).ToString() + " ");
        }
    }
    public static void QuickSort(int[] array, int left, int right)
    {
        if (!(right <= left)) {
            int i = ((left + right) / 2);
            int j = Partition(array, left, right, i);
            QuickSort(array, left, (j - 1));
            QuickSort(array, (j + 1), right);
        }
    }
    private static int Partition(int[] array, int left, int right, int pivotInde
    {
        int i = array[pivotIndex];
        Swap(array, pivotIndex, right);
        int j = left;
        for (int k = left; (k < right); k = (k + 1)) {
            if (!(array[k] > i)) {
                Swap(array, j, k);
                j = (j + 1);
            }
        }
        Swap(array, right, j);
        return j;
    }
    private static void Swap(int[] array, int index1, int index2)
    {
        int i = array[index1];
        array[index1] = array[index2];
        array[index2] = i;
```

Original source code

Here is the original source code for reference.

```
static class QuickSortProgram
{
    public static void Main(string[] args)
    {
        int[] intArray = new int[args.Length];
        for (int i = 0; i < intArray.Length; i++) {
            intArray[i] = int.Parse(args[i]);
        }
        QuickSort(intArray, 0, intArray.Length - 1);
        for (int i = 0; i < intArray.Length; i++) {
            System.Console.Write(intArray[i].ToString() + " ");
        }
    }
    public static void QuickSort(int[] array, int left, int right)
    {
        if (right > left) {
            int pivotIndex = (left + right) / 2;
            int pivotNew = Partition(array, left, right, pivotIndex);
            QuickSort(array, left, pivotNew - 1);
            QuickSort(array, pivotNew + 1, right);
        }
    }
    static int Partition(int[] array, int left, int right, int pivotIndex)
    {
        int pivotValue = array[pivotIndex];
        Swap(array, pivotIndex, right);
        int storeIndex = left;
        for(int i = left; i < right; i++) {
            if (array[i] <= pivotValue) {
                Swap(array, storeIndex, i);
                storeIndex = storeIndex + 1;
            }
        }
        Swap(array, right, storeIndex);
        return storeIndex;
    }
    static void Swap(int[] array, int index1, int index2)
    {
        int tmp = array[index1];
        array[index1] = array[index2];
        array[index2] = tmp;
```

Unexpected difficulties

The *CodeDom* library that I have initially intended to use to output source code in arbitrary *.NET* language has turned out to be quite incomplete. That is, since the library aims to be able to represent source code for any language, it has feature set limited to the lowest common denominator. Therefore, I have switched to *NRefactory* library which is specifically designed with *C#* and *VB.NET* in mind.

Using *T1-T2* transformations for loop finding turned out to be a slightly more difficult since the algorithm is, after all, originally intended to produce a yes or no answer to whether the graph is reducible. However, it was not problematic to refactor the idea to suit a different purpose.

Summary

The project tasks were performed in the planned order and the project is progressing according to the schedule.

The quality of decompilation of the Quick-Sort algorithm is almost 'as good as it gets' so I intend to look for some more complex assembly to tackle.