Michael Ten-Pow
Princeton University
COS Junior Independent Work
Fall 2005

MSIL to JavaScript Compiler

Abstract

The MSIL to JavaScript compiler is a program that translates MSIL programs into equivalent JavaScript programs can be executed within a browser. This is important because of the new found interesting in using JavaScript to develop AJAX applications that target the browser. The problem is that JavaScript development tools are not suited for building large-scale AJAX applications. The MSIL to JavaScript compiler is an effort to bolster JavaScript development efficiency and ease by allowing developers to use the rich set of development tools available for the .NET platform, and having the resulting program run as a JavaScript/AJAX application. MSIL is the intermediary "byte-code" language used by the .NET platform.

Using the MSIL to JavaScript compiler, AJAX developers can take advantage of:

- optimizations such as common subexpression elimination, copy propagation, constant propagation, loop unrolling, etc;
- better debugging tools for the .NET platform;

- strong-typing; weak-typing is also supported;
- better code modularity;
- Intellisense/code auto-completion;
- overall better IDE support.

Though not implemented in this version, the MSIL to JavaScript compiler also lends itself to the following future enhancements:

- code obfuscation;
- dynamic code loading;
- threading and concurrency;
- reflection/introspection.

Part of this project is also a base class library (OSCorlib.dll) that serves as a substitute for Microsoft's mscorlib.dll and defines the basic types usable in our system and their functionality. This document assumes a basic understanding of programming, the .NET Framework, and compiler techniques.

Table of Contents

I. Introduction

As of late, many developers have been using JavaScript to build “AJAX” (Asynchronous JavaScript and XML) applications targeting the browser. AJAX employs a unique client/server model. Applications use JavaScript to interact with the browser DOM (Document Object Model), CSS (Cascading Style Sheets) and various other APIs, such as SVG (Scalable Vector Graphics), in order to build rich, interactive interfaces. And, they use the XMLHTTPRequest API or other hidden form and iframe techniques in order to communicate with a back end server or database.

1.1 Problem

While the benefits of such a model are obvious and have already been vouched for across the Internet, there is a problem: AJAX development is hard because existing JavaScript development tools do not facilitate building these kinds of applications. The solution described in this report is an MSIL to JavaScript compiler, enabling AJAX/JavaScript applications to be written using .NET(MSIL) development tools.

1.2 Why is AJAX development so hard with existing tools?

There are several key reasons why existing tools making AJAX development difficult:

- Existing JavaScript IDEs lack some important features like: automatic code completion/intellisense, inline expansions, re-factoring, project management, integrated source control;
- JavaScript is weakly typed - It is often hard to find errors;
- No concept of interfaces exists nor does any standard class structure - It is harder to work in team environment;
- Missing some important development features that come for free with other languages and platforms:
  - Build systems - Virtually non-existent in the JavaScript. Some do exists, such as the Dojo build system (footnote). But these really lack even some of the basic functionality present in a build system like Make, Ant, or MSBuild;

- Optimization - Since JavaScript runtimes are implemented as interpreters, JavaScript code is relatively slow. In addition, techniques that programmers use to add to code clarity, such as creating JavaScript “namespaces” or even using standard iterative loops, often incur hidden performance penalties. Even standard code like a for loop is not optimal in JavaScript, since using a technique like Duff’s device (footnote) or loop unrolling (footnote) yields significant performance gains;
- Obfuscation - JavaScript source code is clearly visible to those who execute it. This is relevant to developers of commercial JavaScript/AJAX software who need to protect intellectual property;
- Code modularization/componentization - Being prototype based, JavaScript’s inheritance model is often to loosely structured. The result is that different scripts/libraries resort to their own methods for implementing “classes“ and can rarely cooperate with each other without making significant changes. Often times, these competing class systems can break each other. To make matters worse, since there is no concept of “private” or “protected” access in JavaScript, many libraries rely on closures to “hide” implementation details or private/protected fields and methods. This incurs a serious performance penalty. If this method is not used however, all implementation details are freely accessible and code reliability/security goes down. A standard class system is needed;
- Namespaces - JavaScript “namespaces”, which are more often than not simulated by forming closures, have a hidden performance penalty when accessed because of scope chaining. JavaScript sports no built-in namespace maintenance policy or checking mechanism;
- Conditional compilation;
- Code coverage and program analysis;
- Debugging/Profiling: the existing JavaScript debuggers (Venkman[1], Microsoft Script Debugger[2]), offer basic step over, step in, step out, and watch functionality. But unwinding the stack on demand or implementing custom debugger visualizers (Visual Studio .NET[3]) is not possible. Although the Venkman JavaScript debugger has profiling built in, the implementation is quite basic: a call count, maximum call duration, minimum call duration and total call duration are collected for every function. Features like custom hooks are not available, and no facility exists for memory profiling;
- Documentation features and API browsing;

1 More information available at http://www.mozilla.org/projects/venkman

2 More information available at http://msdn.microsoft.com/library/en-us/sdbug/Html/sdbug_1.asp

3 More information available at http://msdn.microsoft.com/vstudio/

1.3 Why use JavaScript?

Even with these apparent problems, web developers are using JavaScript/AJAX more and more. One key reason for this is JavaScript's ubiquity and accessibility via web: More than 90% of internet users run JavaScript enabled browsers[4]. Web based applications like GMail ™ use JavaScript because it allows users to quickly access the application from most any place with internet access[5].

1.4 Potential applications and benefits

The key benefit offered by the MSIL to JavaScript compiler is the ability to build JavaScript/ AJAX applications more efficiently, and that are more reliable and extensible:

- IDE features like code completion and re-factoring become available;
- Strong typing catches errors at compile time and explain what they are definitively;
- Interfaces help make development in a team setting feasible;
- Important development features available to other platforms can be used in JavaScript/ AJAX development:
    - Build systems - NAnt, MSBuild can be used to organize JavaScript/AJAX projects more methodically;
    - Optimization - Code can be optimized at the low JavaScript level, yet code clarity (at the higher language level) is maintained;
    - Obfuscation - Existing MSIL obfuscation tools, such as Dotfuscator Community Edition[6], can be used to obfuscate assemblies before running them through MSIL to JavaScript compiler;
    - Code modularization/componentization - With a more structured and coherent type model and code structure, separating and reusing libraries can be more effective and standard;
    - Namespaces - Namespaces can be used without incurring the performance penalty of that comes with scope chaining;
    - Conditional compilation;

4 Internet usage statistics available at http://www.thecounter.com

5 http://www.gmail.com

6 More information available at http://www.preemptive.com/products/dotfuscator/

    - Code coverage and program analysis - Existing MSIL program analysis tools can identify potential bugs. Likewise for code coverage can be done;
    - Debugging/Profiling - MSIL debuggers are more mature JavaScript debuggers and offer more powerful features. Also, profiling is available. Even though memory management in the CLR is different from that of a JavaScript runtime, they are both garbage collected environments. So, memory "leaks" and memory-related bottlenecks found in the CLR will likely have an analogue in JavaScript.
    - Documentation and API browsing: MSIL documentation and code browsing features are first class. Many .NET supported languages, like C#, support embedded documentation into the source code via comments. These documentation comments are parsed by the compiler and an XML documentation file is generated that can be transformed into a documentation web page or displayed by another documentation front end tool. Code browsing tools like .NET Reflector and the Visual Studio Object Browser make it easy to navigate through a compiled assembly, viewing types, interfaces, and methods -- all with documentation;
- We can solve cross-browser issues using compilation techniques that do not incur the performance penalties that come with existing JavaScript browser abstraction libraries. For example, C# developers can use conditional compilation to produce multiple versions of a library to work with different browser. Traditional browser abstraction libraries wrap functionality, which has a performance penalty;
- Rather than rewriting JavaScript/AJAX libraries in MSIL, wrappers can be written around existing functionality using a feature of the compiler called Interop;
- Opens door for building more features into JavaScript at a later stage:
    - Dynamic code loading;
    - Threading/concurrency;
    - Reflection/introspection.

1.5 Challenge

There are two key challenges to building an MSIL to JavaScript compiler. Firstly, MSIL and JavaScript do not map one to one. MSIL language features like "unsafe" code (code that allows direct memory manipulation), runtime reflection, attributes (meta-programming), P/Invoke (invocation of native system libraries), delegates, tail calls, direct goto, concurrency, and more, are either hard to implement in JavaScript since it does not support these features directly, or cannot be implemented at all. On the other hand, JavaScript is more dynamic than the CLR. This dynamic functionality had to be exposed to MSIL using compiler tricks.

The second key challenge regards the intricate relationship between Microsoft's .NET CLR (Common Language Runtime), which is the runtime system that executes MSIL code, MSIL itself, and the mscorlib.dll base class library, which is referenced with every MSIL assembly. All three are intricately coordinated to work with the other two, and make some fairly strong assumptions about their behavior in order to work properly (from here on out, .NET, MSIL, and CLR will sometimes refer to a common concept and will be used somewhat interchangeably). A similar base class library, OSCorlib.dll, had to be developed to take the place of mscorlib.dll. It does not replicate the entire functionality of mscorlib.dll since this project does not attempt to support porting of existing .NET applications. OSCorlib.dll does, however, perform the roles of defining the basic types available to developers using the MSIL to JavaScript compiler, and making JavaScript-only language features available as well through special types, indexers (methods accessed with array notation), properties (methods accessed with field notation), attributes and method calls. Also, without OSCorlib.dll, any high level language to MSIL compiler, such as the C# compiler mcs, would not function properly.

1.6 Key Contribution

The key contribution offered by this project is a basis for making a more productive and robust JavaScript/AJAX application development environment. The ability to write an AJAX application in a language like C#, which is meant for large-scale development, is a remedy for a majority of the issues present today with AJAX development. Beyond those issues solved, this project offers room for AJAX development to grow past what is currently possible because of limitations imposed by JavaScript. Concurrent programming is an example of one such feature that would be useful to AJAX developers, but is not capable because of JavaScript's limitations.

2. Executive Summary

In the past year, a new demand has appeared for building AJAX applications using JavaScript. JavaScript's ubiquity and accessibility over the web are two of the key reasons that it is in such high demand. However, JavaScript was not designed for building this kind of application and therefore, AJAX development is slow and difficult. The MSIL to JavaScript compiler solves the problems inherent in JavaScript by allowing JavaScript/AJAX applications to be written in .NET, with C# for example, compiled into MSIL using the standard C# to MSIL compiler, and then compiled into JavaScript using the MSIL to JavaScript compiler. Not only does this allow developers to work in C#, which is more suitable for building large applications, but it opens the door for features beyond what is available in JavaScript, such as concurrent programming. The compiler is by no means complete. At this stage it can compile most programs without error, but exception handling is not supported. The compiler also performs some optimizations such as constant propagation, copy propagation, loop optimization, and dead code elimination. The development experience is much better when working in C# as opposed to JavaScript. The features made available by the Visual Studio .NET 2005 IDE are a significant improvement over the simple syntax highlighting offered in most JavaScript IDEs. Optimizations like loop unrolling yielded significant performance gains.

3. Previous Approaches

The MSIL to JavaScript compiler is unique in concept. As of this writing, no publicly available software serves the same purpose, so there is no real “previous work” to look to. However, from one point of view, the previous approach in this area can be thought of as the current state of JavaScript/AJAX development tools.

There are currently two main bands of JavaScript/AJAX development tools:

1. JavaScript-based frameworks, and
2. Rapid Application Development (RAD) tools.

JavaScript-based frameworks, such as DojoToolkit[7], serve as a general purpose abstraction layer for the common elements found across all JavaScript runtimes. Their key utility is to provide a consistent programming model that can be used across most JavaScript runtime in most

7 More information available at http://www.dojotoolkit.org/

browsers. They also provide a standard class model. However, they fail to solve any of the other inherent issues with JavaScript--for instance, weak typing.

AJAX RAD tools such as TIBCO General Interface, JackBe, and Java Studio Creator, provide drag-and-drop functionality for building AJAX GUI applications quickly. In some cases, entire applications can be built using RAD tools without writing any code. In these cases, the problems posed by JavaScript are solved. However, the decreased flexibility imposed by RAD tools is significant. Developers are confined to the set of RAD components that TIBCO or JackBe provides. A reusable class library, for instance, cannot be developed with a RAD tool. At least, not in away that is different from developing one in any other current JavaScript development tool.

4. New Approach and Methodology

4.1 Components

The MSIL to JavaScript compiler is divided into four main components:

1. Frontend - Converts input MSIL assemblies into an abstraction of code and metadata, which is suitable for manipulation by the compiler core;

2. Compiler core - Converts the code abstraction into a CFG, performs optimizations and code transformations, and finally produces a very high level abstraction of the code and metadata for the backend;

3. OSCorlib.dll library - Defines the set of types available in the API and exposes JavaScript functionality otherwise not available from MSIL;

4. Backend - Converts a high level abstraction of code and metadata into JavaScript.

4.2 Frontend

The frontend of the compiler converts inputted MSIL assemblies into an abstraction of code and metadata called the code model.

4.2.1 Possible Implementations

There were two implementations used during this project. It quickly became apparent that implementing an entire frontend would be a daunting and time consuming task. It would entail parsing the Portable Executable (PE) format, and manually extracting the CIL image. From here, the metadata tables would have to be parsed to generate type and metadata objects for the code model. Finally, the MSIL byte code stream for each method would need to be parsed, and a code abstraction built.

Another option was to use the reflection API built into the .NET framework, which resides in the System.Reflection namespace of mscorlib.dll, to read assemblies and extract metadata. However, the reflection API is somewhat limited, so the metadata provided would not be as full as that obtained by parsing the PE by hand. In addition, the .NET reflection API does not provide any sort of code model. Code is simply abstracted to the MSIL level.

So to save time an application called the Lutz Roeder's .NET Reflector[8], a .NET assembly inspection tool, was used as a frontend. In fact, the code model used in the MSIL to JavaScript compiler is derived from the .NET Reflector's code model.

4.2.2 Implementation

4.2.2.1 IFrontend Interface

All frontends must implement the IFrontend interface in order to be used by the MSIL to JavaScript compiler. The IFrontend interface simply contains two methods: void Initialize(I-CompilerCore core) and IAssembly BuildCodeModel(string assemblyFilePath). The main work of the frontend is contained in the BuildCodeModel method. It accepts the file path of the assembly to open and produces an IAssembly object, which is the root type in the code model. The

8 More information available at http://www.aisto.com/roeder/dotnet/

.NET Reflector-based frontend implementation of BuildCodeModel uses .NET remoting to communicate with a running instance of a .NET Reflector application. The .NET Reflector application has a special add-in written for it that loads the specified assembly and extracts the code model information to be returned. The code model itself is built by .NET Reflector, so the implementation here is fairly trivial.

4.2.2.2 Code Model

The code model is comprised of a useful set of abstractions of MSIL language constructs and metadata concepts. The abstractions of MSIL language constructs are structured in such a way that they closely resemble, if not mirror, the kinds of high level language constructs we encounter in a language like C# or Java. Abstractions exist for: binary expressions (IBinaryExpression); assignment statements(IAssignStatement); method invocations(IMethodInvokeExpression), field and property references (IFieldReferenceExpression and IPropertyReferenceExpression); etc. The code model also contains abstractions for even higher level language constructs like while statements (IWhileStatement) and if statements(IConditionalStatement) that are used by the compiler and backend, not the front end.

The abstractions of metadata concepts help the compiler parse through the structure of an MSIL assembly. Abstractions begin at the assembly level, zoom-in further to modules and types, and end at the method body abstraction. After the method body level, abstractions of language constructs take over. Metadata abstractions also provide crucial information for the compiler's operation like information about custom attributes. Custom attributes, which are annotations written into MSIL but are not "executed", give the compiler pointers as to how a particular method is implemented or whether a method is "intrinsic".

Most of the core structure in the code model was taken directly from .NET Reflector's code model. However because it was limiting in many respects (the code model is offered through a set of interfaces, while the real implementation classes are hidden behind and obfuscated), it was re-implemented and changed.

Figures 4-1 and 4-2 are examples of how the code model is used in the compiler core. The first is during reaching definition analysis and the second during tree pattern matching:

```
foreach (CFGNode n in sortedNodes)
{
    Set<Definition> killSet = killMap[n];
    foreach (IStatement statement in n.BasicBlock.Statements)
    {
        IAssignStatement assignStatement = statement as IAssignStatement;
        if (assignStatement != null)
        {
            IVariableReferenceExpression variableReferenceExpression = assignStatement.Target as IVariableReferenceExpression;
            if (variableReferenceExpression != null)
            {
                foreach (Definition def in allDefinitions[((IVariableReferenceExpression)assignStatement.Target).Variable.Name])
                {
                    if (def.Expression.Equals(assignStatement.Expression))
                    {
                        continue;
                    }
                    killSet.Add(def); ;
                }
            }
        }
    }
    killMap[n] = killSet;
}
```

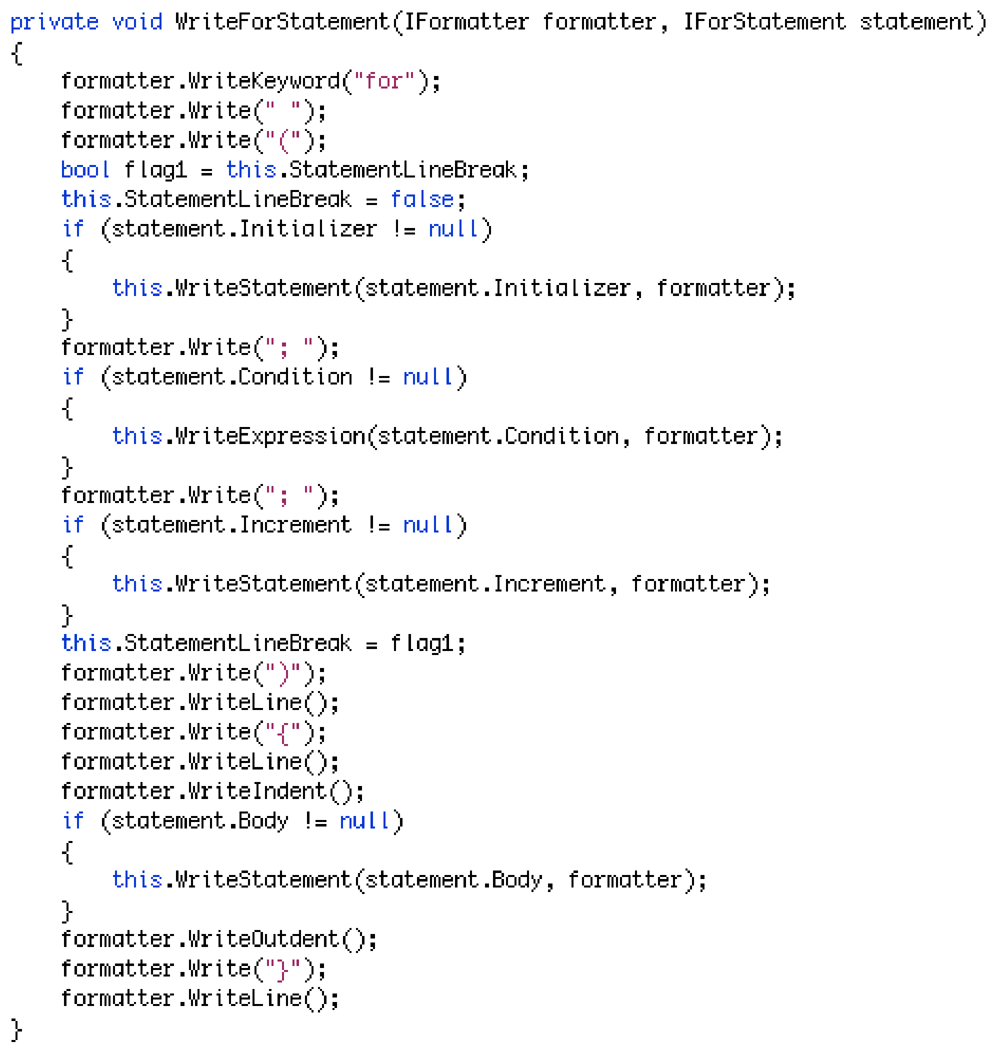
Figure 4-1. CodeGenerator.cs: Lines 8546 - 8569

```
public override object GenerateCode()
{
    if (!IsMatched)
    {
        throw new CompilerException();
    }
    WhileStatement whileStatement = new WhileStatement();
    whileStatement.Condition = (IExpression)Condition.GenerateCode();
    whileStatement.Body = (IBlockStatement)Body.GenerateCode();
    return whileStatement;
}
```

Figure 4-2. WhileStatementPattern.cs: Lines 174 - 184

The first code sample demonstrates the low level aspect of the code model. It contains IStatement, IBlockStatement, IAssignmentStatement and IVariableReferenceExpression objects. The second code sample demonstrates the how the code model is used to create a high level abstraction of code for the backend.

Figure 4-3 is an example of how high level code abstractions in the code model are used in the backend to generate JavaScript code:



```
private void WriteForStatement(IFormatter formatter, IForStatement statement)
{
    formatter.WriteKeyword("for");
    formatter.Write(" ");
    formatter.Write("(");
    bool flag1 = this.StatementLineBreak;
    this.StatementLineBreak = false;
    if (statement.Initializer != null)
    {
        this.WriteStatement(statement.Initializer, formatter);
    }
    formatter.Write("; ");
    if (statement.Condition != null)
    {
        this.WriteExpression(statement.Condition, formatter);
    }
    formatter.Write("; ");
    if (statement.Increment != null)
    {
        this.WriteStatement(statement.Increment, formatter);
    }
    this.StatementLineBreak = flag1;
    formatter.Write(")");
    formatter.WriteLine();
    formatter.Write("{");
    formatter.WriteLine();
    formatter.WriteIndent();
    if (statement.Body != null)
    {
        this.WriteStatement(statement.Body, formatter);
    }
    formatter.WriteOutdent();
    formatter.Write("}");
    formatter.WriteLine();
}
```

Figure 4-3. CodeGenerator.cs: Lines 3377 - 3411

4.2.3 Parts Not Implemented

A custom frontend is partially implemented, but requires much more time to get done. For the purposes of this project, it made more sense to use .NET Reflector as the frontend since the significant innovations are in the compiler core and backend.

4.3 Compiler Core

4.3.1 Implementation

The compiler core accepts a tree of code model objects from the frontend. Specifically, it accepts the IAssembly object generated by the frontend, parses down the object tree to find, IMethodDeclaration objects, and produces CompileInfo objects for each. For each method in an assembly, the corresponding CompileInfo object holds all the context information about the compilation process. Figure 4-4 shows some other implementation details of the CompileInfo type:

```
    public List<CFGEdge> BackEdges { get; set; }
      public Set<CFGNode> DeadCode { get; }
      public DynamicDictionary<CFGNode, Set<string>> Defined { get; }
      public DynamicDictionary<string, Set<CFGNode>> DefinitionsOfVariable { get; }
      public DynamicDictionary<CFGNode, Set<CFGNode>> Dominators { get; }
      public DynamicDictionary<CFGNode, Set<CFGNode>> ForwardOnlyTransitiveClosure {
get; set; }
      public DynamicDictionary<CFGNode, Set<Definition>> GeneratedDefinitions { get; }
      public DynamicDictionary<CFGNode, Set<Definition>> GeneratedExpressions { get; }
      public Dictionary<CFGNode, CFGNode> ImmediateDominator { get; set; }
      public DynamicDictionary<CFGNode, Set<Definition>> KilledDefinitions { get; }
      public DynamicDictionary<CFGNode, Set<Definition>> KilledExpressions { get; }
      public DynamicDictionary<CFGNode, Set<Definition>> LiveDefinitionsIn { get; }
      public DynamicDictionary<CFGNode, Set<Definition>> LiveDefinitionsOut { get; }
      public DynamicDictionary<CFGNode, Set<Definition>> LiveExpressionsIn { get; }
      public DynamicDictionary<CFGNode, Set<Definition>> LiveExpressionsOut { get; }
      public DynamicDictionary<CFGNode, Set<string>> LiveVariablesIn { get; }
      public DynamicDictionary<CFGNode, Set<string>> LiveVariablesOut { get; }
      public Dictionary<string, int> LocalMap { get; }
      public Dictionary<CFGNode, LoopTreeVertex> Loop { get; set; }
      public LoopTree LoopTree { get; set; }
      public int MaxRegisters { get; set; }
      public IMethodDeclaration Method { get; set; }
      public Set<CFGNode> NodeSet { get; set; }
      public Dictionary<string, int> RegisterMap { get; }
      public Dictionary<CFGNode, int> TopologicalOrder { get; set; }
      public List<CFGNode> TopologicalSort { get; set; }
      public DynamicDictionary<CFGNode, Set<CFGNode>> TransitiveClosure { get; set; }
      public DynamicDictionary<CFGNode, Set<string>> Used { get; }
      public DynamicDictionary<string, Set<CFGNode>> UsesOfVariable { get; }
```

Figure 4-4. Part of the public signature of the CompileInfo class

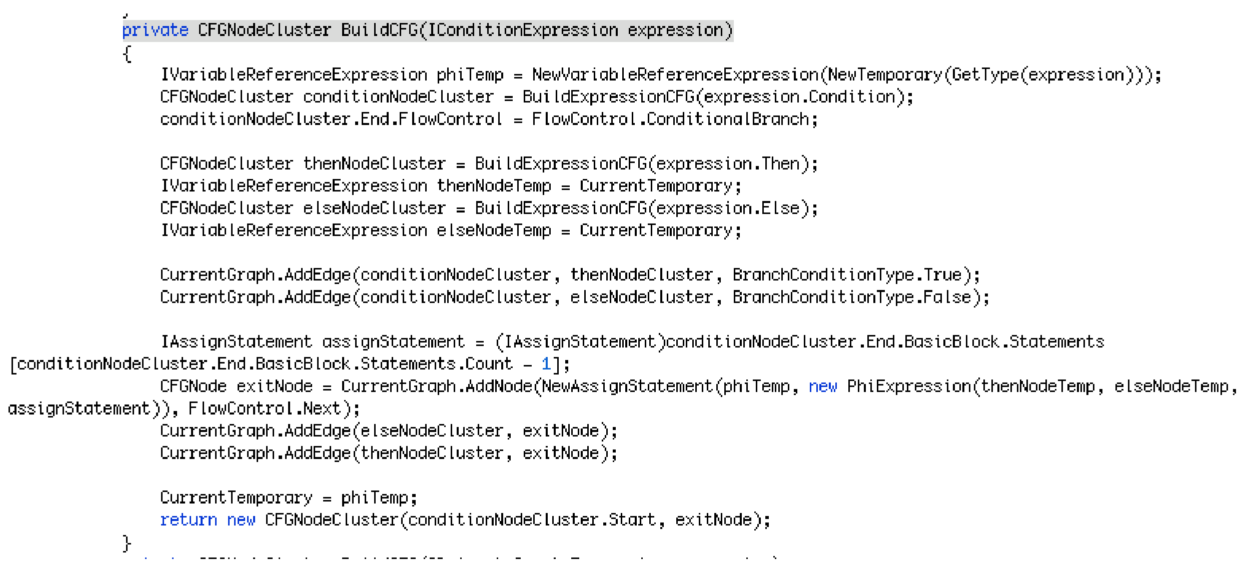
The compiler core then proceeds to perform the following procedures using the CompileInfo object:

- building the CFG;
- program and data-flow analysis;
- optimizations and pseudo-register allocation;
- any available optional preliminary transformations/optimizations;
- pattern tree matching;
- any available optional secondary transformations/optimizations.

Once these steps are completed, the resulting IBlockStatement is stored into the IMethodDeclaration object corresponding to the current method. Once all methods have been processed, the backend is invoked.

4.3.1.1 Building the CFG

Building the CFG entails decomposing the high level code model provided by the .NET Reflector frontend into a set of nodes and edges. Figure 4-5 contains code for building a CFGNodeCluster object, which represents a group of CFG nodes, from an IConditionalStatement code model object:



```
private CFGNodeCluster BuildCFG(IConditionExpression expression)
{
    IVariableReferenceExpression phiTemp = NewVariableReferenceExpression(NewTemporary(GetType(expression)));
    CFGNodeCluster conditionNodeCluster = BuildExpressionCFG(expression.Condition);
    conditionNodeCluster.End.FlowControl = FlowControl.ConditionalBranch;

    CFGNodeCluster thenNodeCluster = BuildExpressionCFG(expression.Then);
    IVariableReferenceExpression thenNodeTemp = CurrentTemporary;
    CFGNodeCluster elseNodeCluster = BuildExpressionCFG(expression.Else);
    IVariableReferenceExpression elseNodeTemp = CurrentTemporary;

    CurrentGraph.AddEdge(conditionNodeCluster, thenNodeCluster, BranchConditionType.True);
    CurrentGraph.AddEdge(conditionNodeCluster, elseNodeCluster, BranchConditionType.False);

    IAssignStatement assignStatement = (IAssignStatement)conditionNodeCluster.End.BasicBlock.Statements
[conditionNodeCluster.End.BasicBlock.Statements.Count - 1];
    CFGNode exitNode = CurrentGraph.AddNode(NewAssignStatement(phiTemp, new PhiExpression(thenNodeTemp, elseNodeTemp,
assignStatement)), FlowControl.Next);
    CurrentGraph.AddEdge(elseNodeCluster, exitNode);
    CurrentGraph.AddEdge(thenNodeCluster, exitNode);

    CurrentTemporary = phiTemp;
    return new CFGNodeCluster(conditionNodeCluster.Start, exitNode);
}
```

Figure 4-5. Building a CFGNodeCluster. CodeGenerator.cs: Lines 9694 - 9715

Figure 4-7 ahead is an example of an entire CFG .

4.3.1.2 Program and Data-flow Analysis

The following, standard program and data-flow analysis operations are performed by the compiler:

- dominator analysis;
- transitive closures;
- loop tree;
- def and use sets;
- liveness analysis;
- reaching definitions;
- available expressions.

The implementations of each operation are straight forward and common components of most compilers. Explaining the significance of each operation here is outside the scope of this document. The results of each operation are stored in the CompileInfo object. Figure 4-6 is an example of the a forward-only CFG, used to compute the transitive closure for each node.

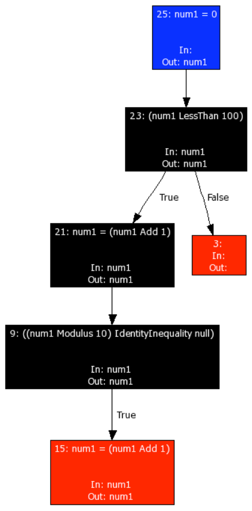


Figure 4-6. A forward-only CFG[9]

9 All control flow graphs pictured in this document were generated using the NGraphViz and QuickGraph for .NET graphing libraries.

4.3.1.3 Optimizations and Pseudo Register Allocation

The compiler core performs several optimizations:

- dead code elimination;
- constant propagation;
- copy propagation;
- common subexpression elimination;
- method inlining;
- loop unrolling with Duff's device[10];
- conversion of nested conditional statements into a switch statement;
- replacing bit shifting binary operations with calls to Math object methods.

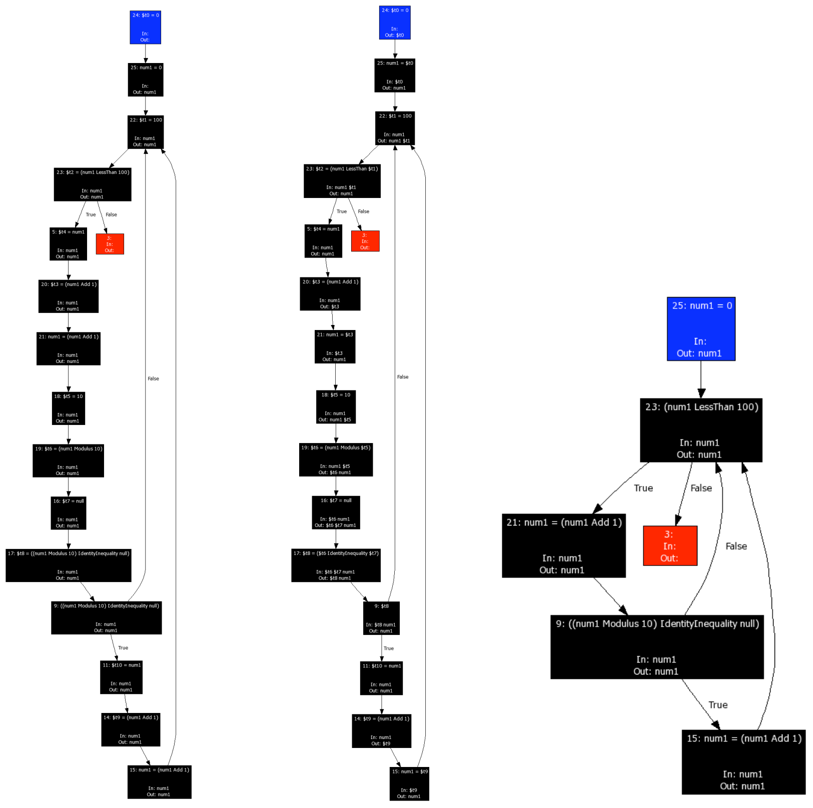


Figure 4-7. Pictured from left to right: Initial CFG; CFG after copy propagation optimization; CFG after dead code elimination.

10 Duff's device is an optimization that allows loops with an arbitrary or even unknown number of iterations to be unrolled. Duff's device is particularly potent in JavaScript.

The implementations of dead code elimination, constant propagation, copy propagation and common subexpression eliminations use standard compilation techniques and the implementation is not specific to the MSIL to JavaScript compiler.

Method inlining works in the following way:

- The MatchCode method (which is searches through arbitrary code model objects using the .NET Reflection API to find and potentially replace code objects according to input parameters) is used to search through each CFGNode's basic block, with parameters set for it to return instances of IMethodInvokeExpression objects.
- At each IMethodInvokeExpression found, a check is made to the compiler core's cache of CompileInfo objects to see if the method has already been processed.
- If it has not, it is processed and placed into the cache.
- MatchCode is again used on the high level code model for the method to determine wether or not it is recursive.
- If the method is recursive, the compiler will not inline this method call.
- The CalculateCost function is called on the method's main block and if the value returned is above the threshold (10), the compiler will not inline this method call.
- If the compiler determines that the method call should be inlined, the CFG for the method to be inlined is grafted into the CFGNode making the call. Sometimes, especially if copy and constant propagation have been run, this will force the compiler to split the CFGNode into two or more nodes and introducing more variables. If this occurs, program and data-flow analysis, optimizations and register allocation need to be run again.
- All method inlining attempts are done in one pass so that the compiler does not have to re-run optimizations, program and data-flow analysis, and register allocation too often.

Although no registers exists in JavaScript, the MSIL to JavaScript compiler treats JavaScript local variables as pseudo-register since access times for variables are fast in JavaScript. Some JavaScript runtimes map a number (unspecified) of local variables to real hardware registers and the rest are spilled or even stored in a normal hash table. So, it is important for the pseudo-register allocator to work as well as possible to stay below that number. The register allocator uses a simple graph coloring algorithm to color nodes in an interference graph. Figure 4-8 depicts the output of pseudo register allocation.

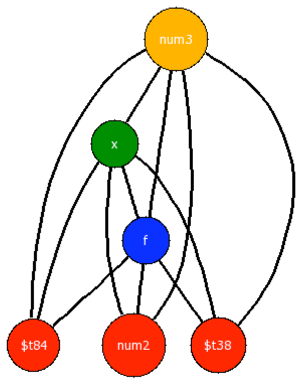


Figure 4-8. An interference graph after register allocation indicates that 4 pseudo-registers need to be used.

4.3.1.4 Optional Transformations/Optimizations

The compiler core allows optional transformations and optimizations to be added. Both must implement the ITransformation interface. The interface allows the optional code to specify when it should be run: before pattern matching, after pattern matching, or both. The Interop feature described in section 5 is implemented as a transformation that occurs after pattern matching.

4.3.1.5 Tree Pattern Matching

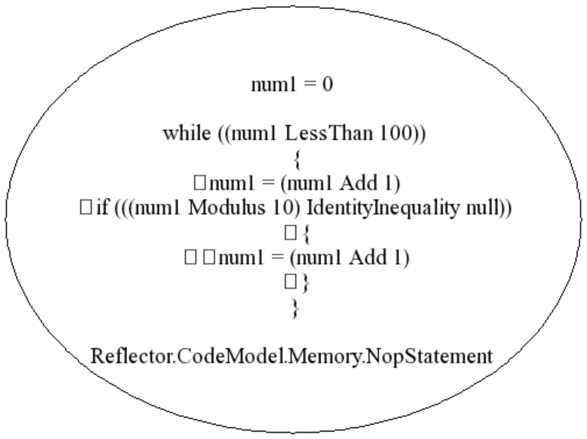
Tree pattern matching is the last mandatory step that the compiler core performs for each method. High level code model abstractions are pattern matched against CFGNodeCluster objects. These high level model abstractions represent constructs like while statements and conditional statements. The pattern matching algorithms are graph based. Figure 4-9 contains part of the implementation for the conditional statement pattern matcher.

```
public override bool Match(CFGNode target)
    {
        IsMatched = false;
        Condition.WorkingSet = WorkingSet;
        Condition.Matched += MatchedHandler;
        MatchedNodes = new Set<CFGNode>();
        if (!Condition.Match(target))
        {
            Condition.Matched -= MatchedHandler;
            return false;
        }
        Condition.Matched -= MatchedHandler;
        int topologicalOrder = -1;
        foreach (CFGNode node in MatchedNodes)
        {
            if (CompileInfo.TopologicalOrder[node] > topologicalOrder)
```

```
                {
                    topologicalOrder = CompileInfo.TopologicalOrder[node];
                }
            }
            if (CompileInfo.TopologicalSort.Count == topologicalOrder)
            {
                return false;
            }
            target = CompileInfo.TopologicalSort[topologicalOrder];

            if (target.Graph.OutDegree(target) != 2)
            {
                return false;
            }
            QuickGraph.Collections.EdgeCollection edges = 
target.Graph.OutEdges(target);
            CFGEdge edge1 = (CFGEdge)edges[0];
            CFGEdge edge2 = (CFGEdge)edges[1];
            if (CompileInfo.TopologicalOrder[edge1.Target] <= 
CompileInfo.TopologicalOrder[edge1.Source] ||
                CompileInfo.TopologicalOrder[edge2.Target] <= 
CompileInfo.TopologicalOrder[edge2.Source]
                )
            {
                return false;
            }
            CFGNode thenNode, elseNode;
            if (edge1.BranchCondition.Type == BranchConditionType.True)
            {
                thenNode = edge1.Target;
                if (edge2.BranchCondition.Type != BranchConditionType.False)
                {
                    return false;
                }
                elseNode = edge2.Target;
            }
            else if (edge2.BranchCondition.Type == BranchConditionType.True)
            {
                thenNode = edge2.Target;
                if (edge1.BranchCondition.Type != BranchConditionType.False)
                {
                    return false;
                }
                elseNode = edge1.Target;
            }
            else
            {
                return false;
            }
            if (!PassesFilter(thenNode) | !PassesFilter(elseNode))
            {
                return false;
            }

            Set<CFGNode> tcThen = 
Filter(CompileInfo.ForwardOnlyTransitiveClosure[thenNode]);
            Set<CFGNode> tcElse = 
Filter(CompileInfo.ForwardOnlyTransitiveClosure[elseNode]);
            Set<CFGNode> thenAndElseOnly = tcThen ^ tcElse;
            Set<CFGNode> sansThenAndElse = tcThen & tcElse;

            Set<CFGNode> thenSet = tcThen - sansThenAndElse;
            Set<CFGNode> elseSet = tcElse - sansThenAndElse;

            if (sansThenAndElse.Count != 0)
            {
                // next node after conditionstatement must dominate everything after 
it
                List<CFGNode> sansThenAndElseList = new List<CFGNode>(sansThenAn-
dElse); // TODO: this algorithm could be better
                sansThenAndElseList.Sort(this.CompareCFGNodesByTopologicalOrder);
                if (sansThenAndElseList.Count > 1 &&
CompileInfo.TopologicalOrder[sansThenAndElseList[0]] >=
CompileInfo.TopologicalOrder[sansThenAndElseList[1]])
                {
                    return false;
                }
            }

            // then node must dominate all nodes in thenSet
            foreach (CFGNode node in thenSet)
            {
                if (!CompileInfo.Dominators[node].Contains(thenNode))
                {
                    return false;
                }
            }
            thenSet.Add(thenNode);

            // same with else node and else set
            foreach (CFGNode node in elseSet)
            {
                if (!CompileInfo.Dominators[node].Contains(elseNode))
                {
                    return false;
                }
            }
            elseSet.Add(elseNode);

            this.Then.WorkingSet = thenSet;
            this.Else.WorkingSet = elseSet;

            this.Then.Matched += MatchedHandler;
            this.Else.Matched += MatchedHandler;

            if (this.Condition.Match(target) && this.Then.Match(thenNode) &&
this.Else.Match(elseNode))
            {
                this.Then.Matched -= MatchedHandler;
                this.Else.Matched -= MatchedHandler;
                this.Condition.Matched -= MatchedHandler;

                OnMatched(new CFGPatternMatch(MatchedNodes));
                IsMatched = true;
                return true;
            }
            else
            {
                this.Then.Matched -= MatchedHandler;
                this.Else.Matched -= MatchedHandler;
                this.Condition.Matched -= MatchedHandler;
                return false;
            }
        }
```

Figure 4-9. Tree pattern matching for a conditional statement. ConditionStatementPattern.cs: Lines 76 - 205

Figure 4-10 contains the result of an entire tree pattern matching operation on a program consisting of a while loop.



```
num1 = 0

while ((num1 LessThan 100))
{
□num1 = (num1 Add 1)
□if (((num1 Modulus 10) IdentityInequality null))
□{
□□num1 = (num1 Add 1)
□}
}

Reflector.CodeModel.Memory.NopStatement
```

Figure 4-10. A CFG (one node) after the pattern matching phase provides a high level abstraction of code for the backend

4.3.2 Parts Not Implemented

4.3.2.1 Exception Handling

Exception handling is not supported by the compiler. Although it was possible to do, the program and data flow analysis for methods with try...catch blocks became useless. So, in order to get to other features, and so that the whole exception handling mechanisms would not have to be re-implemented properly in later versions, exception handling is not supported by the compiler.

4.3.2.2 Threading/Concurrent Programming

Employing a concurrent programming model in AJAX is a desired feature in the development community. The single-threaded approach that JavaScript takes often does not blend well with the asynchronous nature of AJAX. The current paradigm is to use an event driven model utilizing callback functions. Take the following example code, for instance:

```
function retrieve_data(row,col,callback)
{
	var data = check_cache(row,col);
	if(data != null)
	{
		callback(data);
	}
	else
```

```
        {
                // asynchronous method calls callback when completed
                retrieve_data_from_server(row,col,function(retVal)
                {
                        insert_into_cache(row,col,retVal);
                        callback(retVal);
                });
        }
}
```

Figure 4-11. JavaScript code with callbacks for asynchronous calls

We are forced to use a callback here because of retrieve_data_from_server is an asynchronous method. Any code that calls retrieve_data must, must of course, also use a callback to get the data. This situation quickly complicates when multiple calls to asynchronous methods are made in succession and rely upon the previous call. In addition, using callbacks this way forms a new closure, which incurs a performance penalty. A concurrent programming model would allow these asynchronous calls to appear asynchronous. Consider the following C# code:

```
int retrieve_data(int row, int col)
{
        int data = check_cache(row,col);
        if(data != null)
        {
                return data;
        }
        else
        {
                // we can now even make multiple calls without using callbacks
                int newRow = retrieve_newRow_from_server(row,col);
                int newCol = retrieve_newCol_from_server(row,col);
                data = retrieve_data_from_server(newRow,newCol);
                insert_into_cache(row,col,data);
                return data;
        }
}
```

Figure 4-12. C# code that makes asynchronous calls seamless by using threads or continuations.

This clearly shows the benefits of using a concurrent programming model when building AJAX applications. It is possible for the compiler core to implement first continuations and then threads. The JavaScript arguments object could be used to hold the context of a method. The method could be structured in a loop with a switch on an instruction call stored in the context. The switch would jump to the correct section of code depending upon the current value of the instruction pointer.

4.4 OSCorlib.dll

OSCorlib.dll serves to be a replacement for the mscorlib.dll assembly supplied with the .NET Framework. All MSIL assemblies contain references to mscorlib.dll as it provides the basic types made available by the CLR and exposes special features of the CLR such as delegates or P/Invoke. OSCorlib.dll does not attempt to reproduce the functionality of mscorlib.dll. In fact, it cannot since mscorlib.dll is very much tied to the version of the CLR that it runs on. OSCorlib.dll serves the purpose of: 1) declaring the basic types available in JavaScript and exposing them to the MSIL universe; 2) exposing special features of the JavaScript runtime such as functions as objects or expando objects; 3) declaring the types that high level language to MSIL compilers assume are there (otherwise they would fail).

4.4.1 Implementation

4.4.1.1 IntrinsicAttribute

Special types are indicated to the compiler using the IntrinsicAttribute custom attribute. If a type is annotated with the IntrinsicAttribute attribute, the compiler knows not that the implementation for the given type will be present at runtime. Methods for types marked as Intrinsic can declare methods to be extern, in which case the implementation is assumed to be available at runtime. For methods that are not marked extern, the compiler will append code at the end of the outputted JavaScript file that augments the assumed runtime type with the new method. The IntrinsicAttribute attribute also allows the mappings to be specified. For instance, the System.Char type is annotated with [Intrinsic("String", UseGlobalNamespace=true)]. This indicates to the compiler that the System.Char type is really a place holder for some "String" type that will exist at runtime in the global namespace. The IntrinsicAttribute attribute can also be placed on methods. For example, the System.Browser.Window.Alert method is annotated with [Intrinsic("alert",UseGlobalNamespace=true)]. This indicates to the compiler that the static method System.Browser.Window.Alert is really a place holder for some "alert" function that will exist at runtime in the global namespace. If an interface is declared to be Intrinsic, the compiler will modify all invocations of Intrinsic interface methods to use the JavaScript calling convention. The compiler assumes that the interface contract will hold at runtime.

4.4.1.2 Special Types

- System.Object - Intrinsic type mapped to Object
- System.Var - Special type that can be cast to and from anything
- System.String - Intrinsic type mapped to String
- System.Char - Intrinsic type mapped to String
- System.Number - Intrinsic type mapped to Number
- System.Array - Intrinsic type mapped to Array
- System.Void - Dummy type
- System.Boolean - Intrinsic type mapped to Boolean
- System.Enum - Dummy type
- System.Function - Intrinsic type mapped to Function
- System.Delegate - Special type that implements type safe functions using the Function type
- System.MulticastDelegate - same as above
- System.Type - Dummy type
- System.Attribute - Dummy type
- System.Exception - Special Intrinsic type mapping to Exception, which at runtime derives from JavaScript's Error type
- System.EventArgs
- System.ExpandoObject -> Represents the JavaScript Object type. Can be indexed by a String or Number (in hash table manner) just as the original JavaScript Object type can. It was not possible to maintain strong-typing and allow expando (ad-hoc field and method definition) on ExpandoObjects. The indexer solution seemed to serve the purpose of dynamic/expando objects just fine.

  All Intrinsic numeric types map to JavaScript's Number type:
- System.Int64
- System.UInt64
- System.IntPtr
- System.UIntPtr
- System.Byte
- System.SByte
- System.Int16
- System.UInt16
- System.Int32

- System.UInt32
- System.Double

4.4.2 Interop

Interop is a feature of OSCorlib.dll and the compiler that allows existing JavaScript code to be accessed from the MSIL world. It is essentially an extension of the functionality put forth by the IntrinsicAttribute that provides more flexibility. The InteropAttribute attribute is used instead. In addition to what the IntrinsicAttribute attribute allows, the InteropAttribute attribute also enables JavaScript methods and types to be defined in the resource section of the assembly. The InteropAttribute can also take an inline implementation string, which forces a method to be inlined when it is called. The inline implementation string can access parameters to the method using the standard .NET formatting codes "{0}" for the first parameter, "{1}", "{2}" , etc. The "this" object can be referenced with "{this}". The InteropAttribute also allows custom implementation strings for methods and classes, which are similar to inline implementations but do not force method calls to be inlined. In conjunction with the IntrinsicAttribute, Interop is intended to help developers build wrappers for, or simply access, their existing JavaScript code.

4.4.3 Parts Not Implemented

Compatibility with the CLR: Applications targeting the MSIL to JavaScript compiler, but still in MSIL form, will not actually run on Microsoft's CLR because OSCorlib.dll is not compatible with mscorlib.dll. Implementing a compatible OSCorlib.dll would allow the same applications to run both on CLR and in JavaScript. A given application could then be compiled either the JavaScript version or the CLR version of OSCorlib.dll. This would allow debugging and performance testing to be done using MSIL tools, which are more advanced.

4.5 Backend

The backend accepts high level code model objects from the compiler core and prints out the corresponding JavaScript.

4.5.1 Implementation

Although the backend is conceptually the simplest of all four tiers in this project, it holds the most lines of code. For each possible high level language construct outputted by the compiler core, like an IForStatement object, a method exists to translate that type of object into a stream of characters. Figure 4-3 shows the implementation of the WriteForStatement method.

5. Results

5.1 Development Experience

Once OSCorlib.dll was completed, one could right away see the improvement this system offered over traditional JavaScript development. The features provided by the IDE, the documentation features, etc, were all beneficial the development experience. Although there is not much more to report here, this was the real success of the project.

5.2 Optimization Performance Gains

The optimizations, particularly the loop optimizations using Duff's Device, yielded fairly impressive results. Method inlining was by far the worst performing optimization, consistently yielding slower times than non-inlined method calls. Common subexpression elimination, dead code elimination and copy/constant propagation were not really expected to yield significant improvements. They are still key to the compiler because, without them, generated code would be consistently slower than a standard JavaScript implementation because of excess created in the compiler itself. An example of this excess is seen when the CFG is built, since nested expressions are decomposed into a series of definitions to be placed into basic blocks. Each definition creates a new temporary variable.

Figure 5-1 shows average test results after 1000 tests. For each, N is large. All tests were performed on a 867MHZ Powerbook G4 with 768MB of memory in the Mozilla Firefox Browser. Each series of 1000 tests were conducted with a fresh browser window so that memory heap build up would not be a major factor as often is the case with JavaScript. All times are in milliseconds and were recorded using the JavaScript Date functions.

Test	JavaScript	MSIL -> JS
Loops vs. Duffs Device	1095 ms	266 ms
Bit-shifting vs. Math.pow	216 ms	196 ms
Nested conditionals vs. Switch	750 ms	60 ms
Normal local variable use vs. Pseudo register allocation	298 ms	278 ms
Without vs. With method inlining	510 ms	570 ms
Without vs. With CSEE, copy & constant propagation, dead code elimination	768 ms	741 ms

Figure 5-1. Optimization test results.

5.3 Problems with Testing

It was difficult to determine how much of an impact the performance gains would have on real AJAX applications; since, no real AJAX applications were built. Part of the follow up to this project includes building a good test suite.

6. Discussion and Conclusion

6.1 Reflections on Approach

The MSIL to JavaScript compiler introduces a compelling alternative to existing JavaScript/ AJAX development tools. The approach taken seems to be a good one, as building AJAX applications in a language like C# using an IDE like Visual Studio is potentially much more intuitive and productive than using JavaScript. The optimization test results were also encouraging, particularly the loop optimizations. More effort needs to be placed in discovering the right optimization techniques for JavaScript as it is a unique optimization scenario.

6.2 What Was Learned?

Quite a bit was learned during this project--it served as a good extension to CS 320:

- Most everything about the CLR--from how the metadata is structured, to the debugging and profiling APIs, to trade-offs in the JIT compiler, etc;
- Trade-offs in compiler design;
- Difficulties with phase ordering and in performing program/data-flow analysis and optimizations quickly.

6.3 Follow-up Work

Since the results of this project are so encouraging, there is a lot of follow-up work left to be done:

- Thorough testing to find corner cases; Compilers must be perfect;
- Formally show that every possible JavaScript program is available when using the MSIL to JavaScript compiler;
- Support for exception handling;
- Finish implementing a custom frontend. Eliminate reliance upon .NET Reflector;
- More sophisticated optimizations that are proven to have good performance gains in JavaScript;
- Extend the functionality beyond the set of JavaScript programs: Threading/Concurrency with synchronization primitives;
- Obfuscation;
- Writing new frontends and backends to turn the compiler into a basis for a platform. Frontends for Java can be written fairly easily considering how similar MSIL is to Java byte code. Back ends can be written for ActionScript, which has 98% penetration, a greater figure than that of JavaScript;
- Long term possibilities include: a Reflection API, dynamic code emitting, and AOP. These features would entail implementing a runtime system akin to CLR.

7. References

1. Miller, James S., and Susann Ragsdale. The Common Language Infrastructure Annotated Standard. Boston: Addison Wesley, 2003.
2. Standard ECMA-335 CLI. Vers. 3. ECMA. Fall 2005 <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.