MSIL to JavaScript Compiler

Michael Ten-Pow

Problem Description

- ✓ What is it?
- ✓ Why is it important?
- ✓ Why was it hard?

What is it?

- A compiler - translates code into executable programs
  - Input is an MSIL assembly (Microsoft .NET)
    - A program written in C#, VB.NET or another .NET language
  - Output is a functionally equivalent program in JavaScript
    - Runs in a browser environment over the web

Why is it important?

- New interest in JavaScript development
  - AJAX (Asynchronous JavaScript and XML)
  - Web 2.0
- Existing JavaScript development tools are poor - JavaScript was never meant to be used this way
  - No good IDE (Integrated Development Environment)
    - Class outlines, code refactoring, code auto-complete (intellisense), project management
  - JavaScript not strongly-typed
  - Features that come for free with other languages/platforms are not available
    - Build systems, code optimization, code modularization/componentization

Why is it important?

- MSIL has a great set of development tools
  - IDEs: Visual Studio, SharpDevelop, MonoDevelop, X-Develop, Eclipse
- Development can be done in almost any language and compiled to MSIL using existing compilers
  - C#, VB.NET, Java, JScript.NET, C++, OCaml, Boo, IronPython, Perl, and many others
- MSIL gives us several powerful advantages for free
  - Classes, namespaces and other useful language constructs
  - Versioned module system (assemblies)
  - Code optimization
  - XML documentation
  - More…

Why is it important?

- JavaScript is single threaded
  - Asynchronous callbacks - confusing code
  - GUI applications - unresponsive

Why is it hard?

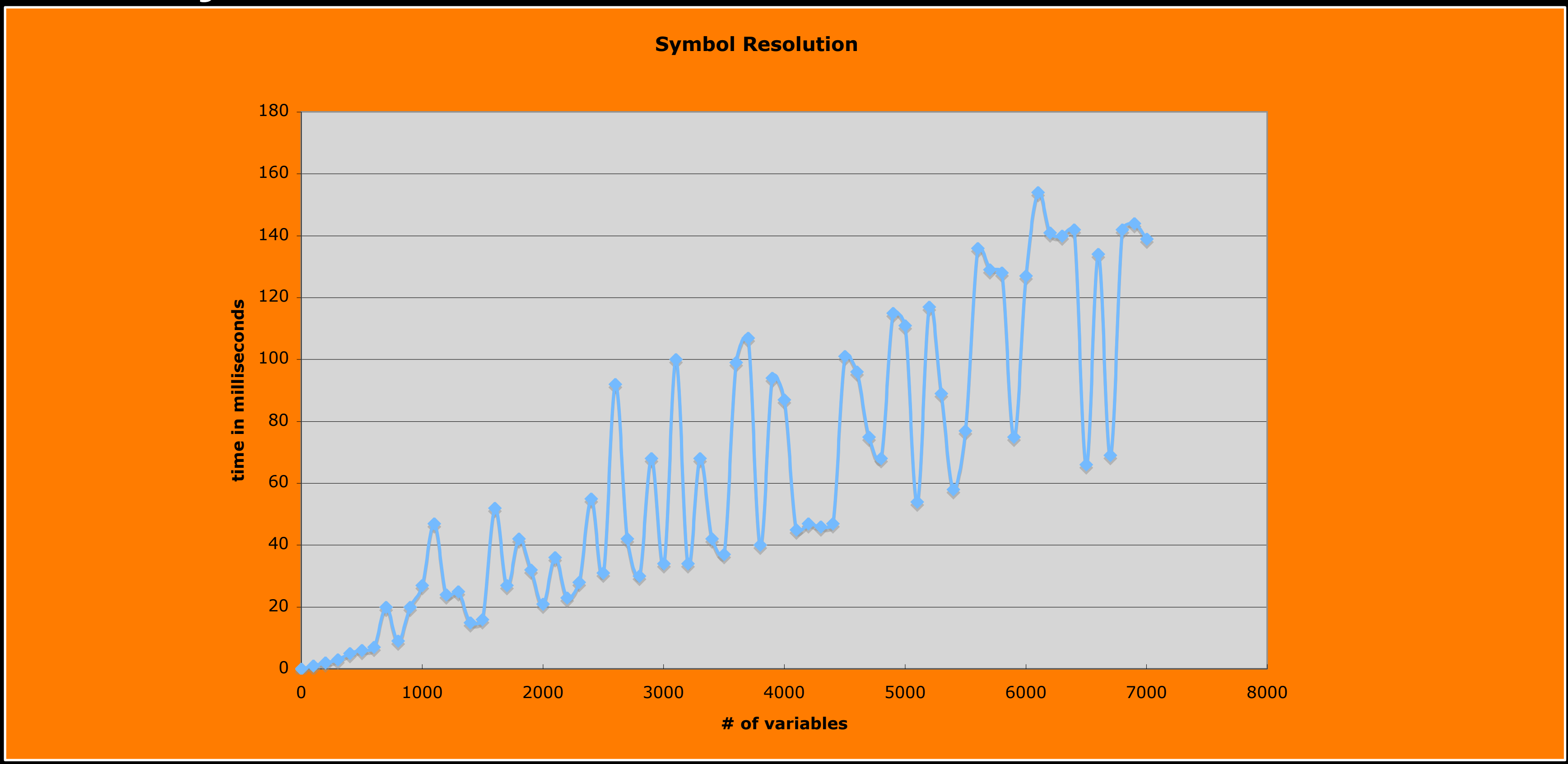
- MSIL and JavaScript do not map “one-to-one”
  - Some MSIL language constructs had to implemented programmatically in JavaScript, or were not supported altogether
  - JavaScript is a very dynamic language, MSIL is more strict. Dynamic aspects of JavaScript are not easily expressed in MSIL. Compiler/API tricks used to capture dynamic nature of JavaScript
- JavaScript is single-threaded
  - Must use “polling” technique to achieve concurrency

Previous Work

- Morfik
- GWT
- Script#
- Disadvantages:
  - No threading!!
  - Symbol resolution
  - Tied to heavy weight frameworks
    - Not Script#

Previous Work

✓ Symbol Resolution



New Approach

- My approach
  - Using existing, mature, well-supported, production quality tools to develop JavaScript applications
  - Support for threading
- Why is this better?
  - Code auto-completion/refactoring in IDE
  - Unit testing
  - Continuous integration
  - No more callbacks
  - GUI applications more responsive

Implementation

- Main parts
  - Compiler
    - Front end
      - Build CFG and code abstractions
    - Middle end
      - Performs optimizations, pseudo-register allocation
    - Back end
      - Code generation (threaded/non-threaded)
    - Linker
      - Resolves symbols and builds executable “binary”
  - Kernel
    - Written entirely in C#, provides runtime for threading
  - Libraries
    - Base class libraries, libraries for DOM, CSS, XmlHttpRequest, etc

Implementation - Front End

- Reads in MSIL assemblies using open source Mono Cecil assembly inspection library
- Builds an abstraction of the code and metadata
  - This is called the “Code Model”
- Front ends can be written to support any other input language
  - There is a clean interface to implement this
  - Java support would be quite easy to implement

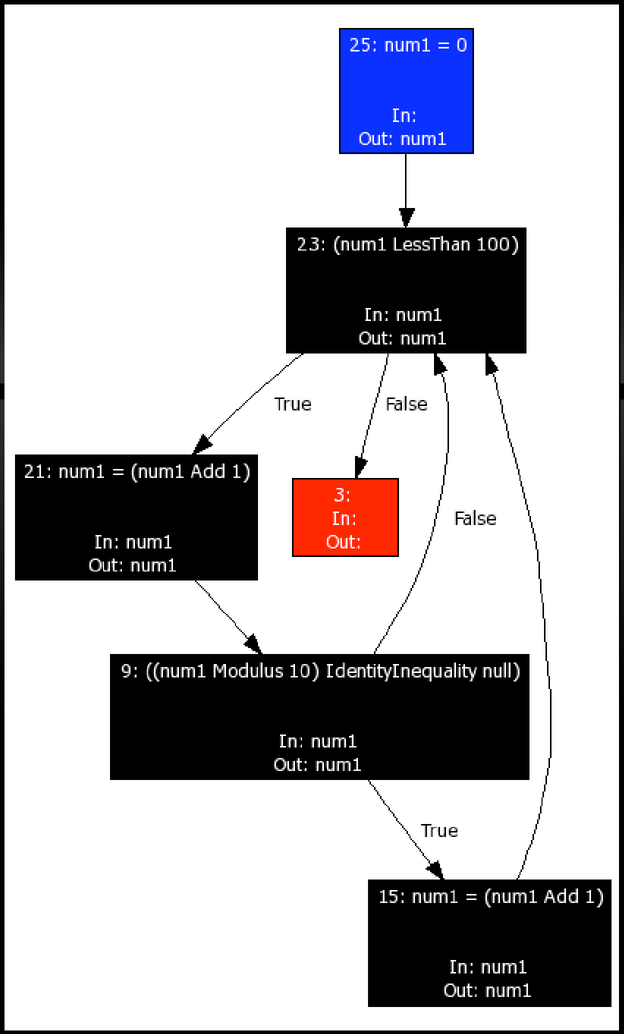
Implementation - Front End

- Code Model:
  - Contains useful abstractions of MSIL language constructs and metadata concepts. For example:
    - IAssembly = MSIL assembly
    - IAssembly -> GetTypes() returns ITypeDeclaration array
    - ITypeDeclaration -> GetMethods() returns IMethodDeclaration array
    - IMethodDeclaration -> MethodBody property:
      - Locals, arguments, code size, custom attributes, etc
      - MSIL code stream in bytes
      - CFG representation of code (most valuable)
        - IAssignStatement
        - IMethodInvokeExpression
        - IBinaryExpression
  - Code model contains on the order of 100 different classes

Implementation - Front End

✓ CFG example:

```
__p.TestLoop = function()
{
    var num1 = 0;
    while(num1 < 100)
    {
        num += 1;
        if(num1 % 10)
        {
            num1 += 1;
        }
    }
}
```



Implementation - Middle end

- ✓ Manipulates and optimizes intermediate form (CFG) to prepare for back end code generation

Implementation - Middle end

- Basic steps (program/data analysis):
  - Separate complex CFGNodes into more simple ones
  - Dominator analysis - which nodes ALWAYS come before other nodes as code is executed?
  - Transitive closures - set of all nodes reachable from a given node
  - Single Static Assignment (SSA) -
  - Loop tree - which loops are inner loops? (useful for optimization)
  - Def and Use sets - which variables are defined and used at a given node?
  - Liveness - which variables are “live” at a given node? (store meaningful data which is used later on)
  - Reaching definitions - what are the possible definitions of a variable at a given node?
  - Gen and Kill sets - like “Liveness” but for expressions
  - Optimizations
  - Register allocation
- Actual steps involve several iterations of these basic steps and in different orders (phase ordering)

Implementation - Middle end

- Optimizations
  - Copy propagation
  - Constant propagation
  - Constant folding
  - Dead code elimination

Implementation - Middle end

- Optimizations example:
  - Demonstrates copy/constant propagation, constant folding, and dead code elimination

Original C# (not optimized):

```
private int TestReachingDefs()
{
        int num1 = 10;
        int num2 = 12;
        num2 = num1;
        num1 = 13;
        return num2;
}
```

Compiler output (optimized):

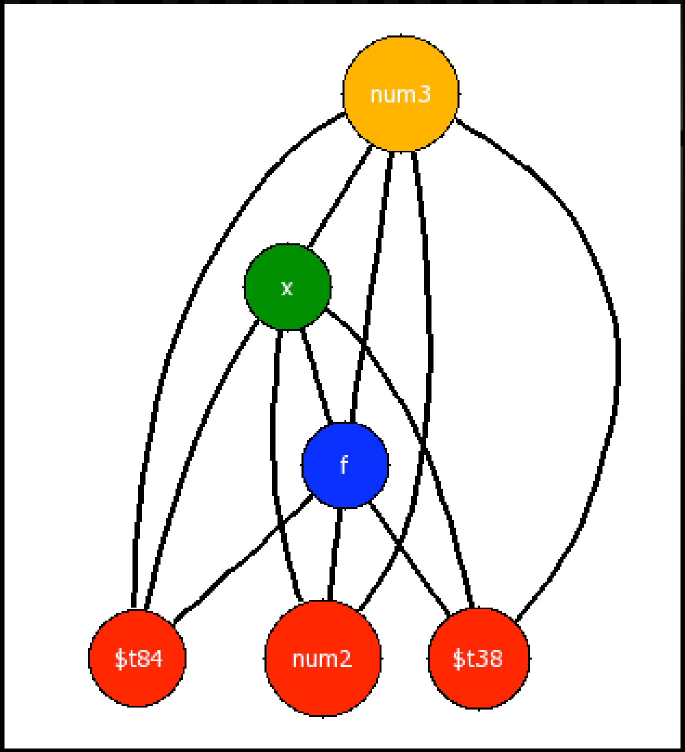
```
__p.TestReachingDefs = function()
{
        return 10;
}
```

Implementation - Middle end

- Pseudo-register allocation
  - Pseudo-registers are JavaScript local variables
  - Register allocation allows us to reuse pseudo-registers that are no longer live
  - In certain JavaScript runtimes (Rhino JS runtime), local variables are mapped to actual machine registers at runtime.
- Graph coloring algorithm

Implementation - Compiler

- Register allocation example:
  - Interference graph
    - $t38 and $t84 are temporary variables
    - x, f, num2 and num3 are real local variables or arguments
    - 6 variables reduced to only 4 pseudo-registers



Implementation - Back end

- Perform code generation
  - Preemptive code
  - Non-preemptive code
- Emits object file for linker
- New back ends can be written to generate code for other runtime systems
  - Actionscript
    - Also runs in browser
    - Adobe claims 98% penetration (more than JavaScript)
    - Hardly any cross-browser issues
    - Flash player 8.5 features JIT compiler
      - Significantly faster than interpreted JavaScript

Implementation - Kernel

- Written entirely in C#
  - Facilitates execution of threaded code
  - Simple priority based scheduler
  - Provides mechanisms for context switching

Implementation - Libraries

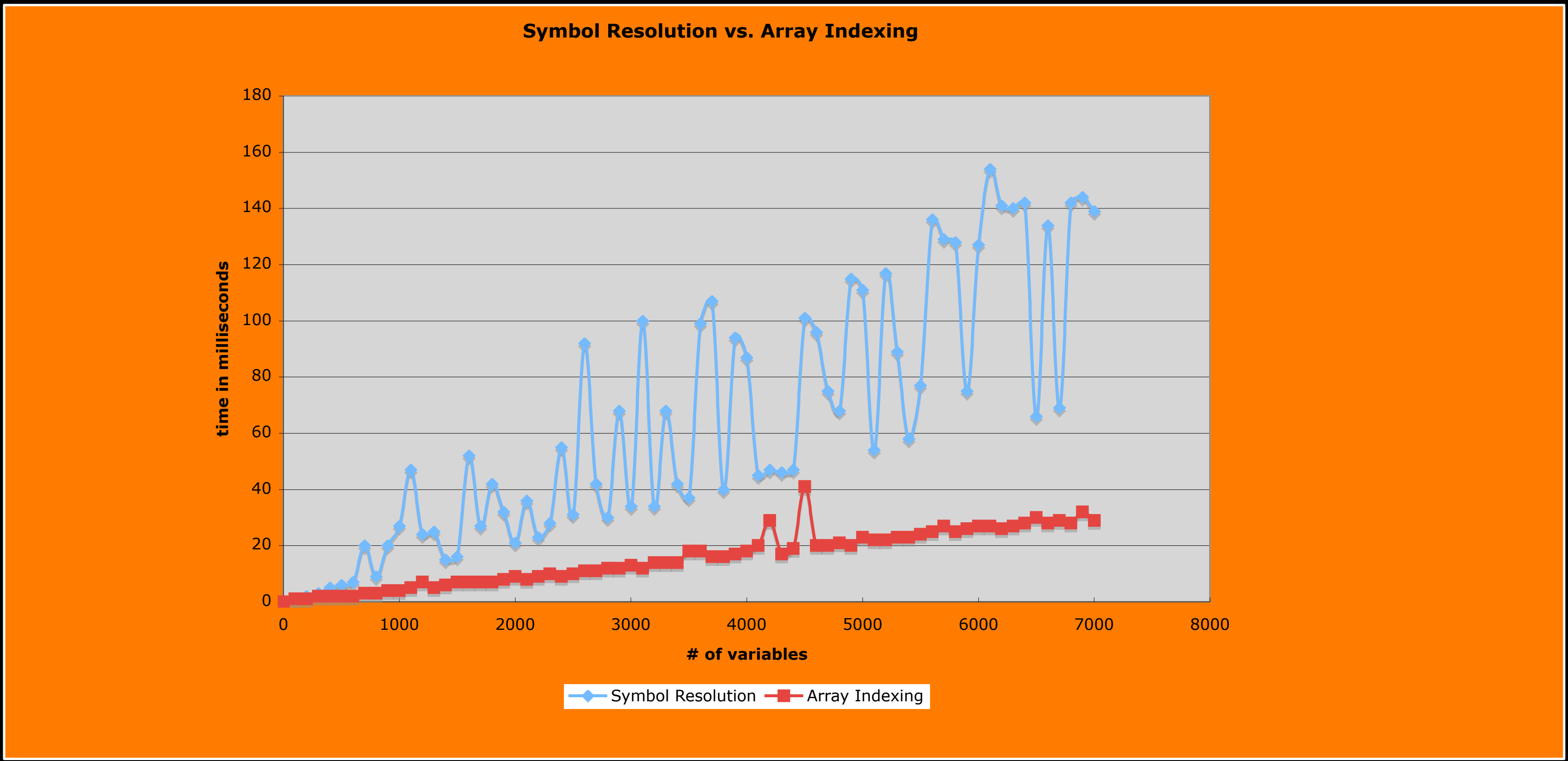
- Base class library (OSCorlib.dll) replacing mscorlib.dll
  - Maps special .NET types to built-in JavaScript types
    - Object, String, Number, Error, etc
  - Provides abstractions for threading
    - Thread
    - Locks
    - Conditions
    - Semaphores
- System.Browser.dll
  - DOM, CSS, XmlHttpRequest interfaces
  - Shows interoperability with existing code

Results

- Threaded code is slower than hand-written JavaScript
  - However, perceived performance is not restricted
  - “This script has been unresponsive...” - no longer an issue
- Scope chains are shortened to a maximum of 2 levels
  - JavaScript programmers modularize code using closures
  - This has hidden impact on performance
  - Compiler flattens scope but maintains namespace coherence
  - Speed increase of by factor of 2 in some cases

Results

✓ No more symbols at runtime



Results

- ✓ Development experience:
  - ✓ Writing C# in Visual Studio is more efficient
  - ✓ Intellisense
  - ✓ Code overview
  - ✓ Documentation *****