

An ECMAScript compiler for the .NET framework.

César López Natarén Elisa Viso Gurovich

Facultad de Ciencias
Universidad Nacional Autónoma de México

Work sponsored by Novell Inc.

30 de septiembre de 2005

- ECMAScript programming language.
- Common Language Infrastructure.
- Mono's ECMAScript compiler.
- First-class functions, dynamic objects, late binding and nested functions implementation.

ECMAScript programming language

- Designed by Brendan Eich at Netscape.
- Better known as JavaScript or JScript.

ECMAScript's design philosophy

- Scripting language.
- Designed to be:
 - Flexible.
 - Powerful.
 - Easy to learn.

ECMAScript's features

- Object oriented.
- Make use of prototypes for state and behavior inheritance.
- Dynamically typed.
- First class functions.

ECMAScript's objects

- An ECMAScript object is an unordered set of properties.
- New instances are constructed through constructors.
- We have no classes.
- Prototypes to the rescue!

ECMAScript's prototypes

- We inherit from a common parent:
 - State
 - Behavior
- We can add our special state and behavior, too.

ECMAScript's prototypes sample

```
var x = new Array (1, 3, 4);
```

```
try {  
    print (x.DoesNotExists ());  
} catch (e) {  
    ;  
}
```

```
Array.prototype.DoesNotExists =  
    function (x) { print (x); };
```

```
print (x.DoesNotExists ('Hello ENC-2005'));
```

- Type annotations are not included.
- Very difficult to perform strong typing at compilation time.
- Give us the flexibility of:
 - Using undeclared variables.
 - Assign values to undeclared properties.
- We need a way to find/create the things at runtime.

ECMAScript's functions

- They are objects.
- They are expressions.
- In general, they are first class citizens:
 - Can be stored in data structures.
 - Can be passed as arguments to other functions.

Common Language Infrastructure

- Let's call it CLI.
- ECMA/ISO international standard.
- Specifies a virtual execution system.

What does it form the CLI?

- Common Type System (CTS).
 - Rich type system design for programming languages interoperability.
 - Reference types and value types.
- Common Language Specification (CLS).
 - Set of rules that increase the interoperability.
- Metadata.
 - Neutral.
 - Used to describe value and reference types.
- Common Intermediate Language (CIL).
 - Set of operation codes.
- Virtual Execution System (VES).
 - Enforces the CTS rules.
 - Provides the support required to execute the CIL.

Mono's JScript compiler

- The work presented on the paper.
- This work is sponsored by Novell Inc.
- Let's call it mjs.
- Written in the C# programming language.

- Typical architecture:
 - Lexical analysis.
 - Syntatic analysis
 - Semantic analysis.
 - Code generation.

- Semantic analyzer:
 - Two phases, PopulateContext and Resolve ops.
 - Bind identifiers to nearest declarations.
 - Late bound analysis of access to properties, index access and method calls.

- Code generator:
 - We use the System.Reflection.Emit API.
 - Code generator maps:
 - One class per file.
 - Each class has a method 'Global Code' which contains each of the global operations at the file.
 - Generate an appropriate entry point for each of the classes.
 - Two phases.
 - Global Code first.
 - Recursively generate methods bodies.

We need first-class functions, dynamic objects, late binding and nested scopes

- There're a couple of problems to be solved.
 - No direct first class functions on the CLI.
 - Objects are static on the CLI.
 - Everything is strong typed on the CLI.
 - Lack of notion of nested methods in the CLI.

First-class functions

- Let's remember that:
 - Fields and local variables can be passed as arguments.
 - Fields and local variables can be the return value of a function.
- What we do for first-class functions is:
 - If we encounter a global function declaration, we create a field of type `ScriptFunction`. When we encounter a nested function declaration we create a local variable of that type.
 - We build its closure in the body of the container method.
 - Assign the closure to the field or local variable.
 - Everytime the method name is referenced we pass the field or local variable which is of type `ScriptFunction` and knows how to perform a real method invocation in case is needed.

First class functions sample

```
function F ()
{
  function Square (y)
  {
    return y * y;
  }
  return Square;
}

print (F () (2));
```

First class functions on CIL

```
.class 'JScript 0' {  
  .field ScriptFunction F  
  .method 'Global Code' {  
    call Closure JScriptFunctionDeclaration (...)  
    stsfld F  
  }  
  .method F {  
    .locals (ScriptFunction G)  
    call Closure JScriptFunctionDeclaration (...)  
    stloc G  
    ldloc G  
    ret  
  }  
}
```

- We solve this problem creating an embedded object system.
- Each ECMAScript object implements the IExpando interface.
 - AddField, AddMethod, AddProperty, ...
- Use delegates when adding methods.

- Semantic analysis and code generation for binary expressions that involve the dot and index access operators are treated in a special manner.
- Why?
- We can't assert that some property is/isn't part of an object/variable at compilation time, which is the reason we must generate code that at runtime make the proper checks.
- Also, being able to add new properties to an object on the fly is an ECMAScript requirement.
- Except for the static properties of the built-in objects.

Late binding

- A LateBinding class stores the left and right side of the expressions.
- Every time we want to perform an assignment, we use LateBinding's SetValue method.
 - SetValue verifies that the left hand side is an object.
 - SetValue stores the given value for future use.
- In reading access situations we use LateBinding's Call method.
 - Receives an array of arguments.
 - Receives info that let's it know if it's in a bracket situation and if it's a constructor.
- Both methods use the Reflection services in order to find the needed data.

ECMAScript late binding sample

```
var x = new Object ();  
x.y = new RegExp (/a|ab/);  
print (x.y.exec ('`abc`'));
```

Late binding on the CIL

```
.class 'JScript 0' {  
  .method 'Global Code' {  
    .locals (LateBinding V_0)  
    ldstr ``y``  
    newobj LateBinding::.ctor (string)  
    stloc V_0  
    ldsfld 'JScript 0'::x  
    stfld LateBinding::obj  
    ldloc V_0  
    ldstr ``a|ab``  
    call RegExpConstructor::Construct (...)  
    call LateBinding::SetValue (...)  
  }  
}
```

The nested method problem

- We can have function declarations/expressions inside another function declaration/expression.
- Variables declared in an outer scope can be used in the nested scope.
- In the CLI we do not have the notion of nested methods.
- We are not allowed to reference variables, methods generated in a different code generator (ILGenerator).
- We need to propagate the changes.

Propagating the state

- We solve this problem:
 - Generating code that explicitly creates a stack frame that contains the variables defined in the outer scope.
 - Transform a direct `OpCodes.Call` invocation to `CallValue`, as the id bound to a method name may change on a nested scope.
 - Copy back the possible changes that were done.

Nested function sample

```
function f ()
{
  var x = ``set at f``;

  function g ()
  {
    x = ``set in g``;
  }

  g ();

  print (``x = `` , x);
}

f ();
```

Nested functions on CIL

```
.method f { .locals (object x)
  call VsaEngine::ScriptObjectStackTop
  castclass StackFrame
  ldfld StackFrame::localVars
  dup
  ldc.i4 0
  ldloc.0
  stelem.ref
  LateBinding::CallValue
  call VsaEngine::ScriptObjectStackTop
  castclass StackFrame
  dup
  ldc.i4.0
  ldelem.ref
  stloc.0
}
```

Nested functions on CIL (2)

```
.method G { .locals (object V_0)
  ldstr ``set in g''
  call VsaEngine::ScriptObjectStackTop ()
  call ScriptObject::GetParent ()
  castclass StackFrame
  ldfld StackFrame::localVars
  dup
  ldc.i4 0
  ldelem.ref
  stloc.0
  stloc.0
```

Nested functions on CIL (3)

```
    call VsaEngine::ScriptObjectStackTop ()
    call ScriptObject::GetParent ()
    dup
    castclass StackFrame
    dup
    ldc.i4.0
    ldloc.0
    stelem.ref
  }
}
```

Conclusions

- It's possible to implement dynamic languages on the CLI.
- We are currently passing 5053 out of 6478 compilable tests from the Mozilla/JavaScript test suite.

- Implement closures, first class methods at metadata/CIL level.
- Add continuations support to mjs.
- Add E4X to mjs.

- Project page: <http://mono-project.com/JScript>
- These slides:
<http://lambda.fciencias.unam.mx/~cesar/mjs.pdf>
- Email: cesar@ciencias.unam.mx
- Novell Inc. <http://www.novell.com>

Thank you very much

- Questions?