

# Compiling Ruby on the CLR

Wayne Kelly

Faculty of Information Technology  
Queensland University of Technology  
Queensland, Australia  
+61 7 3138 9336

w.kelly@qut.edu.au

John Gough

Faculty of Information Technology  
Queensland University of Technology  
Queensland, Australia  
+61 7 3138 9334

j.gough@qut.edu.au

## ABSTRACT

The implementation of statically typed programming languages on the .NET CLR is by now well understood [1]. However, the situation with dynamic languages is not so clear. Typically such languages have objects that are dynamically typed, while the CLR is statically typed at the instruction code level. Nevertheless there is a growing body of evidence suggesting that the CLR can be a suitable target for such languages [2]. In order to better understand the issues involved we set out to create a full implementation of the Ruby language on the CLR. This paper describes the challenges faced and design decisions made in creating Ruby.NET – a Ruby compiler for the CLR.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors – Code generation, Compilers, Interpreters, Optimization, Parsing, Run-time environments and compiler generators.

## General Terms

Design, Languages and Performance

## Keywords

Dynamic languages, Ruby, CLR, .NET and Compilers.

## 1. INTRODUCTION

Our Gardens Point Ruby.NET compiler is not just a Ruby/.NET bridge, nor a Ruby Interpreter implemented on .NET, but a true .NET compiler. The compiler can be used to statically compile Ruby source files into a fully verifiable CLR v2.0 assembly or it can be used to directly execute a Ruby source file (compile, load and execute). Our implementation consists of a Ruby runtime library (`RubyRuntime.dll`) that contains an implementation of Ruby's built-in classes and modules. The Runtime library also contains our compiler infrastructure as Ruby's dynamic semantics require us to be able to perform further compilation at runtime.

We provide two front-ends to our compiler. One called `RubyCompiler.exe` is designed to compile a set of Ruby source files into a .NET assembly (either `.exe` or `.dll`). It takes similar command line arguments as Microsoft's C# compiler and is designed to be used by our Visual Studio integration package which allows Visual Studio users to develop Ruby.NET projects. Our other front-end called `Ruby.exe` takes approximately the

same command line arguments as the standard Ruby 1.8.4 interpreter and is designed to mimic its behavior by dynamically compiling, loading and executing a single Ruby source file. The code generated by both of our front-ends is able to load additional Ruby files which may be either pre-compiled dynamic link libraries or plain Ruby source files which need to be compiled on the fly. This file loading process uses Ruby's standard mechanisms for locating other files to load rather than using .NET assembly probing mechanisms.

## 1.1 Formal Specification

The first challenge that we faced was the lack of a formal specification for the Ruby language. This means that the most widely used implementation of Ruby – an interpreter implemented in C [3,4] becomes the *de facto* specification. Fortunately, that implementation is open source and the license allows others to derive new works from it. We henceforth refer to that implementation of the Ruby language as *C-Ruby*.

## 1.2 Parsing

Parsing modern programming languages is normally a relatively straight forward task. Unfortunately, Ruby's syntax is derived from Perl which is notoriously difficult to parse using traditional techniques. On first inspection the C-Ruby grammar appears unnecessarily complex. For example, the syntax for array indexing is replicated in five separate contexts. C-Ruby's YACC based parser is also tightly coupled to the hand written scanner with the two working closely together to resolve many potential ambiguities. We made extensive efforts to simplify the grammar and scanner but were ultimately defeated by the complexity and the lack of formal techniques for reasoning about the equivalence of different grammars.

Ultimately, we simply mirrored the implementation used in C-Ruby. Even this was not a straight forward exercise as there were no reliable YACC like tools for the CLR. We were therefore forced to create our own tool – the Gardens Point Parser Generator (GPPG) [5]. As we found, it was critical for our tool to behave virtually identically to YACC (or Bison). For example, when contemplating a reduce operation, the next input character should only be read if it is necessary to resolve between ambiguous reduce operations. This subtle behavior would go unnoticed in parsing most languages – but it was necessary for Ruby due to the tight interaction between the parser and scanner.

### 1.3 Compiling vs Interpreting

Ruby has traditionally been implemented as an interpreter. At “runtime” – the interpreter parses the Ruby source code, builds a tree data structure that represents the program, and then walks over this tree interpreting each node as it goes. Interpretation generally consists of two steps. The interpreter must first inspect the current tree node to *determine* what to do next, and then it must actually *perform* the appropriate operation.

A compiler does the parsing and determining what needs to be done at compile time. Compiled code therefore *generally* runs faster because at runtime because it only needs to *perform* the appropriate operations. Dynamic languages such as Ruby, however, introduce the possibility of new source code being constructed and encountered for the first time at runtime. In order to “compile” such languages we need the compiler infrastructure to be present at runtime so that we can dynamically compile and load the new code.

The dynamic semantics of languages such as Ruby also diminish the traditional performance advantages of compilation. Take for example the operation of adding two integers together. In a strongly typed language, the compiler will know at compile time that an integer addition is required and will be able to generate very efficient code. In Ruby, however, when we come across a “+” operator at compile time we cannot determine that it will be an integer addition as the *types* of the operands in the expression cannot be determined in general. Even if we did know the type of one of the operands, for example if the left operand was an integer literal, then we still couldn’t be certain that an integer addition was required as someone may have overridden the “+” method for the *Fixnum* class. This overriding may have taken place in a separately compiled component, so we have no way of knowing whether this might have happened.

Now let us assume that the “+” method for the *Fixnum* class has not been overridden. The standard implementation for this method must still inspect the type of the right operand before we can determine the type of addition that is required. If the type of both operands is found to be integer then even then we can not simply add them together. We must check that the addition does not result in an overflow; otherwise we will need to promote the operands to *Bignum*.

As this example illustrates, the cost of the interpretation step tends to be relatively small compared to all the other tests that must be performed at runtime to achieve the correct dynamic semantics. The relative performance advantage of compilation is therefore diminished.

In the case of compiling for the CLR we face two further performance impediments compared to the native C-interpreter. Firstly, the compiled CIL code that we generate must be further just-in-time compiled at runtime into native code. Secondly our goal is to generate fully managed and verifiable CIL code. This means that the code we generate must be completely type safe, so we can not do the kind of pointer manipulation and type conversion tricks that the C-interpreter uses.

So, in summary, we believe it is possible to implement Ruby in a fully compiled manner, but we do not necessarily expect it to run significantly faster than the C-Ruby interpreter. The principal

advantage then of producing a .NET compiler for Ruby is not improved performance, but providing the benefits of the CLR platform to Ruby programmers and to add Ruby to the set of languages that .NET programmers can choose from.

## 2. MAPPING RUBY TO THE CLR

Ruby has objects, classes and methods. The CLR also supports objects, classes and methods so one might think compiling Ruby to the CLR to be a relatively straight forward exercise as it is with languages such as C#. However, as the previous section highlights, Ruby’s dynamic semantics hinders such straight forward implementations.

Firstly, the good news - all Ruby objects belong to a class and the class that an object belongs to cannot change. The super-class from which a class inherits can also not change. The bad news is that the set of instance variables of an object and the set of methods belonging to a class can change at runtime. Variables and expressions are also not typed. So when we invoke a method, we generally do not statically know the type of the receiver object. Even in those cases where the type of the receiver can be inferred, we still don’t statically know anything about the method that will bind to that method name at runtime – we don’t even know how many parameters it will expect.

### 2.1 Ruby Classes

CLR classes have a fixed set of methods, so we cannot use CLR classes to represent Ruby classes. While we could dynamically generate CIL code for a Ruby class at runtime, once we have created a CLR class and created instances of it – it is then impossible to add or modify the set of methods that it supports. Ruby allows classes to be modified after instances of that class have been created. Such changes to a Ruby class can also occur in separately compiled source files, so it is generally impossible to have complete knowledge of a Ruby class at compile time. We therefore provide a CLR class called *Ruby.Class* to represent Ruby classes at runtime. These class objects contain a reference to their super-class (another *Ruby.Class* object) and a table that maps method names to the methods to which they are currently bound.

The process then of invoking a Ruby method is to:

- 1) Determine the Ruby class of the receiver object.
- 2) Determine the Ruby method currently bound to the specified method name for that Ruby class.
- 3) Invoke the found Ruby method.

### 2.2 Ruby Methods

Ruby methods need to be represented in such a way that they can be referenced in method tables. We could use CLR delegates for this purpose but since Ruby methods are not statically associated with a class, we choose to instead create a separate CLR class for each Ruby method. We use a singleton pattern to ensure only one instance of each Ruby method class is created – both for efficiency and for identity purposes.

These Ruby method classes contain a *Call* method which is used for invoking them. As these methods do not belong to the class of

the receiver object, the receiver (or `self`) must be passed in as an explicit parameter. These compiler implementation details are of course completely hidden from application programmers.

If we had chosen to represent Ruby methods as delegates we would have had to choose a standard signature which would have involved passing an array of arguments. We avoid the overhead of allocating and initializing such an array in most common cases by providing overloaded `Call` methods catering for each number of arguments up to 10, plus a general purpose `Call` method for greater than 10 arguments. Each concrete Ruby method class overrides the `Call` method corresponding to the number of arguments that it expects. The abstract `RubyMethod` base class automatically takes care of calls to that method with other numbers of arguments.

## 2.3 Ruby Objects

All Ruby objects inherit from a Ruby class called `Object` which implements a number of standard methods such as `class`, `clone`, `freeze`, `inspect` and `methods`. An obvious approach therefore would be to require all objects used within Ruby programs to derive from a CLR class or interface called `Ruby.Object`. We do provide such a class but we do not assume that all objects passed as parameters to Ruby methods (or stored in Ruby collection classes) are derived from `Ruby.Object`. We instead allow any CLR reference type (derived from `System.Object`) to be used.

This has two advantages. Firstly, it supports interoperability with other .NET languages, without having to wrap other CLR objects within Ruby objects. Secondly, it allows us to represent primitive Ruby types more efficiently. For example, we can represent a Ruby object of class `Fixnum` as a (boxed) `Int32`, Ruby objects of class `TrueClass` and `FalseClass` can be represented as boxed `Bools` and Ruby objects of class `NilClass` are simply represented as a null reference.

So, if the only thing we know about an object is that it derives from `System.Object`, how can we usefully operate on it? What does a Ruby object need to be able to do?

## 2.4 Finding an Object's Ruby Class

To invoke a method on an object we first need to be able to determine its Ruby class. Rather than relying on the object itself to provide a field or method that returns its Ruby class, we instead provide a static method that takes a `System.Object` and returns its `Ruby.Class`. In most cases, the objects we use will derive from `Ruby.Object` – in which case the static method simply returns the `Ruby.Class` object stored in the `Ruby.Object`. If the object is `null`, a boxed `bool` or a boxed `int32` then the method return the `Ruby.Class` object corresponding to `NilClass`, `TrueClass`, `FalseClass` or `Fixnum` class, as appropriate. Otherwise the object must have been created by a component implemented in another .NET language. In that case, we dynamically create a special `Ruby.Class` object to represent the foreign CLR type. We maintain a static table that maps foreign CLR types to their corresponding `Ruby.Class` object, so that if we encounter an object of that type again, we can use the already created

`Ruby.Class` object rather than creating a new one. We use a special subclass of `Ruby.Class` called `Ruby.CLRClass` that uses CLR reflection to locate methods rather than the method table used by other `Ruby.Class` objects.

## 2.5 Instance Variables

In statically typed languages, the class defines the set of instance variable (or fields) that a class may possess, but the value of those instance variables is a property of each individual object. In Ruby, the class does not define a fixed set of instance variables. Each object of a particular class may have a different set of instance variables, and the set of instance variables associated with an object can change dynamically – separately compiled components may independently add to the set of instance variables that an object possesses. So, in general, the only way of representing instance variables is as a dictionary that maps instance variable names to their current value. And since instance variables are not typed, those values must simply be a reference to an object derived from `System.Object`.

It is therefore possible for different Ruby objects belonging to different Ruby classes to have the same CLR class (say `Ruby.Object`). These `Ruby.Object` objects will each contain references to their respective `Ruby.Class` objects and will each contain a field for their own instance variable dictionary. But what about `Fixnums`, `TrueClass`, etc? Their representations are not derived from `Ruby.Object`; where do their instance variables get stored? We use the same trick as C-Ruby and store their instance variables in two dimensional look-aside table indexed by object and instance variable name.

## 2.6 Allocating Ruby Objects

As the previous section suggests, apart from a few special cases such as `int32s` and `bools`, we can represent most Ruby objects using just the `Ruby.Object` CLR class. There are two reasons why we would not always want to do this. Firstly, the Ruby language defines a set of built-in Ruby classes such as `String`, `Array`, `Hash`, `Regexp`, `File`, `Dir` etc. Objects of these classes have an *intrinsic* value. For example a `String` encapsulates a list of characters. We therefore provide in our runtime library, separate CLR classes (derived from `Ruby.Object`) for each of the Ruby built-in classes. Each of these classes contains a private `value` field that is used by in our implementation of the methods for these classes. For example the `Ruby.String` class contains a `value` field of type `System.String`. We cannot directly represent Ruby strings as CLR strings as CLR strings are immutable whereas Ruby strings are mutable. The `Ruby.String` class contains methods to for example capitalize or reverse the string. These methods work by changing the `value` field of the `Ruby.String` object.

Obviously, then if a Ruby program calls the standard `new` method of the `String` class, we need to allocate a CLR object of class `Ruby.String` rather than just a `Ruby.Object`. If however, the Ruby program calls the `new` method of a class `Foo` that inherits from `String`, then we also need to ensure that we allocate a CLR object that at least derives from `Ruby.String`. The problem is that we cannot generally determine at compile time the super-class of a Ruby class. The

super-class of a Ruby class is not allowed to change after the class has first been declared, but the super-class may be a runtime expression rather than being a statically known value. Even when the super-class is seemingly static, all may not be as it appears. Consider the following example:

```
class Foo < String
  ...
end
```

It might appear from this example that we know that class `Foo` inherits from the built-in `String` class. However, the super-class expression `String` is really just a “constant” expression, and in theory, the programmer could have previously redefined this constant to refer to some other `String` class – perhaps their own custom `String` class. This redefinition could have occurred in a separately compiled component, so in general, we have no way of knowing for certain what the base class of a Ruby class is at compile time.

When we are asked to create an instance of class `Foo`, we do not necessarily have to create an instance of a CLR class called `Foo`, but if `Foo` does actually inherit from built-in class `String` then we need to ensure that we at least allocate an object that derives from `Ruby.String` (so that the implementation of the standard string methods can access its value field). Equally, if it turns out `String` is actually an alias for say built-in class `File`, then we need to at least ensure that we allocate an object that derives from class `Ruby.File`. The decision of which class to allocate must therefore be made at runtime. We achieve this by using Ruby's normal method binding mechanisms. We invoke a Ruby method called `allocate` to create the appropriate class of object. In this case, class `Foo` might not possess an `allocate` method of its own, but one of its superclasses will. If the parent-class turns out to be `String` (or derived from `String`), then the `String` `allocate` method will be encountered before the `Object` `allocate` method using Ruby's normal inheritance hierarchy search process.

The other reason we may not want to simply use an object of CLR class `Ruby.Object` to represent a Ruby `Foo` object is for interoperability purposes. Using a `Ruby.Object` object will achieve all of the appropriate semantics for a program implemented entirely in Ruby. But if we wish components implemented in other .NET languages to be able to conveniently use our Ruby classes, it is preferable if we create a CLR class specifically for Ruby class `Foo` and allocate such an object when required. But again, the problem is knowing statically what the base class is. So, in the above example, we would create a CLR class called `Foo` that inherits from `Ruby.String`, but at runtime we would have to decide whether the `allocator` for class `Foo`, ie the `allocator` that creates an instance of CLR class `Foo` was actually *safe* to use. If that `allocator` creates an object that does not derive from the type of object created by the base classes' `allocator` then we instead defer to using the base classes' `allocator`.

In either event, the CLR class `Foo` that we create is primarily a wrapper class that simply provides convenient access to that Ruby classes' methods and instance variables. We add CLR methods to this class corresponding to each of the statically known Ruby methods in the class, but these wrapper methods simply invoke

the underlying Ruby methods via the normal dynamic Ruby method lookup process. In this way, the implementation of those methods can still change dynamically. The Ruby object may also gain additional dynamic methods that are not accessible via the static CLR wrapper methods but they can still be called using a more dynamic invoke API.

## 2.7 Local Variables

While the set of instance variables associated with an object is impossible to determine statically, the set of local variables used within a method or block is generally known at compile time. Each invocation of a method effectively gets its own copy of each of these variables. Such local variables would in less dynamic languages be allocated on the runtime stack as their lifetime corresponds to the duration of the corresponding method call. In Ruby, however, local variables may continue to live after the method that created them has returned. This can occur when a Ruby code block is created within that method. The code within the block has access to all of the local variables in its surrounding method. Such blocks can be treated as objects and be returned or otherwise escape the scope of the method. If this happens, the captured local variables need to remain live for as long as the block may be executed.

So, in general, rather than storing Ruby local variables as CLR local variables (which are stored on the runtime stack), we store Ruby local variables in special activation frame objects that we allocate on the CLR heap. At compile time we can generally determine which local variables will be used within each method, so we create a separate activation frame class (containing named fields for each local variable) for each Ruby method and block. The prologue of the `Call` method in each Ruby method class is responsible for allocating a new activation frame object. A reference to this frame object is stored in a CLR local variable so that it can be conveniently accessed in the remainder of the method. So, to access a local variable, we need to follow the CLR local variable to the activation frame and then access a named field within that activation frame. This is actually more efficient than the C-Ruby implementation that traverses a list of local variables to find a match. If a nested block exists and escapes from the method then the block will contain a reference to the frame. The frame object will remain live and then naturally be garbage collected as soon as there are no longer any references to it.

## 2.8 Blocks

Ruby code blocks can also be nested inside other code blocks. Inner code blocks therefore have access to all local variables in surrounding blocks as well as the surrounding method. Blocks can be invoked just like Ruby methods so we represent each block by a class derived from `RubyMethod`. These custom created `Block` classes contain fields which point to each of its surrounding activation frame objects. When we access a local variable from within a nested block we always know statically at which nesting level it was defined. So to access such a local variable, we simply need to access the block's (strongly typed) field that corresponds to that level and then access the appropriately named local variable field within that activation frame.

## 2.9 Passing Parameters to Blocks

Blocks behave much like regular methods in that they both provide a list of formal arguments and they can be encapsulated in a *Proc* object and then be either called or invoked by `yield`. The processing of assigning actual parameters to formal parameters for blocks does however differ from the process used for methods. The assignment of actual parameters to formals for blocks is basically treated the same as Ruby's parallel assignment construct. So, for example, calling the block

```
{ |x, y, z| ... }
```

with arguments `expr1`, `expr2`, `expr3` is equivalent to:

```
x, y, z = expr1, expr2, expr3
```

This however means that we can have “strange” formal argument lists such as:

```
{ |a, (b, *c), c[a]| ... }
```

which contains duplicate arguments and L-value expressions.

Another important property of parallel assignment is that the syntax of the right hand side affects the semantics, not just its value. For example:

```
a, *b = [1,2,3]
```

produces different results to

```
a, *b = [1,2,3], *[]
```

even though their right hand sides have effectively the same value. A consequence of this is that if you pass these two different right hand sides to a method then the semantics will be the same, but if you pass them to a block via a `yield` command, then the semantics will be different (and if you pass them to a block via a `call` then the semantics are the same!)

This mean, when evaluating arguments lists, we need to maintain, not just the list of values computed, but also a flag indicating whether the syntax consisted of a single right hand side argument.

## 2.10 Dynamic Evaluation

Ruby provides a runtime method that takes a dynamically constructed string and interprets it as Ruby source code at runtime. In the simplest case, this is not difficult for us to achieve. We simply invoke our compiler at runtime and rather than writing the compiled code to a file, we write it to a memory stream, dynamically load the assembly and invoke the generated code. We use our own PE file writer for all code generation. Our writer is based on the published binary format specification used for CIL assembly files and avoids the verification steps performed by other CIL emitting APIs (verification is still performed by the CLR when the memory stream is loaded as an assembly).

The more complicated aspect of implementing the `eval` method is providing access to Ruby classes and local variables that already exist at runtime in outer contexts within which the `eval` method is invoked. Our implementation of the `eval` method first uses CLR reflection to determine the context within which it is invoked. It then sets up a static compiler context (abstract syntax tree) that encapsulates this runtime context. The given string is then parsed within this synthetically generated compile time

context, so that all local variables encountered during parsing are automatically mapped to local variables and frame types that already exist at runtime.

The runtime context provided to the `eval` method might be the current runtime context or it may be a different runtime context as captured previously and encapsulated in a Ruby *Binding* object. A *Binding* object encapsulates the current `self` or receiver object as well as the current activation frame.

## 2.11 Dynamic Local Variables

Normally, as stated earlier, the set of local variables for a method or block can be statically determined. However, this is not true of local variables created by calls to the `eval` method. These dynamic local variables can, however, only be accessed by other calls to `eval` within the same frame. Other static code within the frame will not know of these local variables and will treat them as an undefined local or method. Consider for example:

```
def foo
  # x created dynamically within eval
  eval 'x = 42'

  # x is not treated as a local here
  puts x # undefined local or method

  # but x is visible here inside an eval
  eval ' bar {|y| puts x }'
end
```

Such dynamic local variables need to be accessed via a dictionary (similar to instance variables). But, as these dynamic local variables can only arise within `eval` code – they are relatively rare and we can lazily create a dictionary for them only if it turns out one is actually needed.

## 2.12 Non-local Control flow

One of the principal uses of code blocks in Ruby is as the body of a `for` or `each` loop. In this context, control flow constructs such as `break`, `retry`, `redo` and `return` make sense. `Break` leaves the block and continues after the loop, `redo` goes back to the start of the block, `retry` goes back to the start of the loop and `return` leaves the entire method. It must be remembered, however, that a block may escape from the method in which it is defined. If a control flow statement is executed in such a situation then the resulting control flow can be very non-local.

Consider the Ruby code below. A block is created within `methodA`. It is saved as a *Proc* object and later passed to `methodB` and subsequently passed as a block parameter to `methodC`. The “`yield`” in `methodC` executes the block. The `return` statement causes control to return not just from the block or the method that invoked `yield`, but all the way back to return from the method in which the block was declared (if it is still active).

```

def methodA()
  # create block
  x = proc { return 42; };
  # and then much later ...
  methodB(x);
  puts 'end methodA';
end

def methodB(y)
  methodC(&y);
  puts 'end methodB';
end

def methodC()
  yield();
  puts 'end methodC';
end

methodA();

```

In this case the block was declared within *methodA*, so control will leave the block, skip the end of *methodC*, skip the end of *methodB* and return from *methodA*. If the *return* statement is replaced by a *break* statement, then control will instead return to the end of *methodA*. A *retry* would cause *methodB* to re-execute (causing an infinite loop in this case).

Note: this non-local branching behavior only occurs if the block is invoked by *yield*. If we instead passed *y* as a *Proc* parameter to *methodC* and then called it, the *return* statement would only return control from the block back to *methodC*. The behavior also depends on how the block was defined, as *lambda* blocks behave differently to *proc* blocks and differently again if you pass a method as a *Proc*. All together there are about 24 semantic cases to consider – many of which are non-orthogonal and seem counter-intuitive. We assume many of these cases are simply consequences of the current C-Ruby implementation rather than carefully considered language design choices.

In any event, our basic approach to achieving this kind of non-local branching behavior is to use CLR exceptions. We use three distinct exception classes for *return*, *retry* and *break*. If a *redo*, *break* or *retry* occurs within a loop then an appropriate branch instruction is generated. In contexts where non-local branching is required we instead throw an exception of the appropriate kind (*return*, *retry* or *break*). Each block has a reference to the frame of the scope in which it was defined. This defining scope is stored in the exception object when it is thrown, together with the any return value that may be provided by the control flow statement. We generate code that places every Ruby method call within its own *try* block which catches *Break* and *Retry* Exceptions. When one of these exceptions is caught, we check to see if the defining scope stored in the exception matches the current frame. If it does, then the appropriate frame has been found, so it branches to the appropriate label within that method and uses the return value

stored in the exception. If the frames do not match, then the exception is re-thrown to the next outer call level. *Return* exceptions are caught by *try* blocks that we generate around each scope (class init, method or body) rather than the *try* blocks which surround each method call.

This process is potentially quite time consuming but it is only needed when such non-local branching constructs are encountered at runtime. In most cases control will return from a method call normally and none of the code in these catch blocks will ever be executed. This code pattern does however seriously increase the amount of code that we generate as it is replicated at each call site.

## 2.13 Continuations

Continuations allow a snapshot of the runtime stack to be made and to then return to that state at some latter point. We have not yet implemented continuations as there is still much discussion going on regarding whether they will be retained in future versions of the Ruby language. The CLR does not provide developers sufficient access to the CLR stack to perform these kinds of operations directly, so our intended approach for dealing with continuations also relies on CLR exceptions.

When a continuation is to be created, we would throw a special CLR continuation exception. The *try* block around each Ruby method call would also contain a special catch clause for this kind of exception. This catch block would save all of its local state (including the call point) into the continuation object and then re-throw the exception to the next call level for it to do the same. When this exception reaches the outermost level, we will have by that time captured all the state information needed for the continuation but unfortunately in the process completely pulled down the runtime stack. So, to be able to continue normally after having created the continuation, we then need to go about the process of restoring the CLR call stack to its original state.

Each method would have a special recreating parameter flag and the prologue code of each method would depend on this flag either execute the method normally, or jump to the call point where execution of this stack frame was previously. When a continuation is called, the same process would be used to pull down the current call stack and to re-establish the stack of the continuation.

This approach relies on all of the methods on the CLR call stack being constructed according to a strict pattern to support the tearing down and building up of call stacks. This is fine for methods generated by our compiler. The hundreds of methods of Ruby's built-in classes and modules that we have implemented by hand would be more problematic. It does not work at all if there are methods on the stack from components implemented using other .NET languages.

## 2.14 Rescue Clauses

Ruby supports the raising of exceptions. These exceptions all inherit from a built-in Ruby class called *Exception* which we represent using a CLR class called *Ruby.Exception*. Since *Ruby.Exception* inherits from *Ruby.Object* and not *System.Exception*, it is necessary to use a separate CLR class called *RubyException* (which does inherit from

`System.Exception`) to actually throw the exception. The `Ruby.Exception raise` method generates a `RubyException` object and the two objects work together, with each containing a reference to the other.

Ruby also supports `rescue` clauses for catching Ruby exceptions. Each `rescue` clause contains a list of *exceptions* that it will catch. This exception list, however, is not necessarily a list of the names of the exception classes. In general, it is a list of expressions which are meant to evaluate to Ruby objects that support method `"==="`. This Ruby method (which may be programmer supplied) is used to determine whether the current exception matches the specified exception. So, rather than just catching a particular CLR exception based on the Ruby exception listed in the `rescue` clause, we must effectively catch all Ruby exceptions and then check to see if it was one we were meant to catch, and if not, re-throw it. For interoperability purposes, we also want to be able to catch arbitrary other CLR exceptions in Ruby `rescue` clauses by wrapping them in Ruby exceptions. Rather than trying to catch and convert CLR exceptions to Ruby exceptions as close to their origin as possible, we instead lazily only convert them when and if they are caught by a `rescue` clause. If a CLR exception is thrown that is not within a `rescue` clause then it remains as a pure CLR exception. We therefore need to catch *all* CLR exceptions at every `rescue` clause (not just those derived from `RubyException`). We must therefore be careful, to ensure that we do not trap CLR exceptions such as our `Break` exceptions that we use for the implementation of non-local control flow.

Ruby also supports a separate and slightly different exception mechanism based on `throw` and `catch` constructs, but that mechanism is implemented as part of the built-in class library rather than as a language feature. We use a separate CLR exception class to implement that type of exception.

## 2.15 Ruby Threads

The Ruby language defines its own threading model rather than relying on the threading model of the underlying platform. Ruby's thread model is often described as a "Green" thread model as it does not support the concurrent execution of its threads. The standard Ruby interpreter uses a single operating system thread and manually time slices between them at certain designated places within the runtime. Each thread is guaranteed at least 10ms of uninterrupted execution before possibly being switched out.

We have not yet implemented Ruby threads as there is still some debate regarding whether their semantics will change in future versions of the Ruby language. Our plan, however, for implementing them was to use separate CLR threads for each Ruby thread, but to carefully control the use of those threads to ensure that only one thread was executing at any given time and that switches between threads could only happen at designed places within the runtime.

We have, however, also tried to make our implementation as thread safe as possible, so that we can also support concurrent CLR threads. We have avoided using global variables to represent quantities that should be specific to the currently executing thread. For example, we don't store the current class in a global variable as the C-Ruby interpreter does.

## 2.16 Code Generation Invariants

As explained in the previous sections, CLR exceptions are used as part of the Ruby.NET implementation to achieve non-local control flow. This means that all Ruby method calls need to be placed in their own `try-catch` block. One of the CLR's verification rules is that the CLR argument stack must be empty when entering a `try` block. This poses problems when method calls are nested. Consider for example:

```
expr1.Foo(expr2, expr3.Bar(expr4, expr5));
```

Normally this would be translated into:

```
Code for expr1
Code for expr2
{
Code for expr3
Code for expr4
Code for expr5
Call Bar
}
Call Foo
```

But if we put a `try` block around just the call to `Bar`, then the stack would contain `expr1` and `expr2` on entry to the block.

We therefore need to translate it into:

```
temp1 = Code for expr1;
temp2 = Code for expr2;
temp3 = Code for expr3;
temp4 = Code for expr4;
temp5 = Code for expr5;
try {
    load temp3
    load temp4
    load temp5
    temp6 = Call Bar
} catch ...
try {
    load temp1
    load temp2
    load temp6
    Call Foo
} catch ...
```

Obviously if any of the expressions are literal expressions or expressions that do not themselves involve method calls, then we can simply load that literal rather than storing it in a temp. Our code generation for method calls is then based around the idea of *pre-computing* each of the arguments. If the argument is a literal then the pre-computing step is a no-op, otherwise pre-computing

computes the value and stores it in a temp. These temps are recycled once we are finished with them. This is a very important pattern as just about everything in Ruby is done via a method call. For example applying the '+' operator is a method call, determining whether a Ruby exception matches a particular Ruby `rescue` clause is a method call, etc.

## 2.17 Different Notions of Current Class

The Ruby language has at least three different notions of what the “current class” is in a given context.

One definition denoted `ruby_cbase` represents the current lexical class and is used for accessing constants and class variables. Another definition denoted `ruby_class` is used when defining and undefining methods and aliasing. This notion of current class may not be the same as the current lexical class if an `eval` method is in progress that is using a `Binding` from another context. The dynamic nature of this notion of current class requires us to propagate it via a parameter passed to all blocks. The third notion of current class, denoted `last_class` is used for making super calls. It is propagated via a parameter passed to every Ruby method and used by our `FindSuperMethod` method to ensure that when a super method is called, that we look for a method higher up the class hierarchy than where the currently executing method was defined. Without it, an infinite loop can result when invoking super methods.

## 2.18 Object Ids and Weak References

Ruby Objects support a method called `id` which maps to a unique integer associated with that object. The C-Ruby interpreter basically just returns the address of the object in question. It is not possible to do this on the CLR in a strongly typed, safe and verifiable manner. We don't however, need to return the address of the object, we can return any integer, provided it uniquely corresponds to this object. It is a simple process to simply generate the next integer id in sequence when a new object asks for its `id`. Once we have returned an `id` we need to store it with the object so that we can ensure we will return the same `id` if asked again.

The bigger challenge is implemented the reverse operation that locates a Ruby object given its `id`. This can be done by maintaining a reverse lookup table. The potential problem with this approach is that placing an object in this table will prevent it from being garbage collected. The solution is to use a special CLR class called `System.WeakReference` which maintains a reference to an object without preventing it from being garbage collected. We can query a `WeakReference` object at any time to determine if the object that it points to has been garbage collected. We use this same approach for implementing the `each_object` method of the build-in module `ObjectSpaces` which enumerates all of the currently live Ruby objects.

## 2.19 Dynamic Code Loading

We may need to dynamically generate CIL code in two circumstances, firstly for `eval` methods and secondly when another Ruby source file is loaded. If the source file in question has already been compiled into a dynamic link library and that

library is newer than both the source file and the Ruby compiler itself, then we load the precompiled library and use it. Otherwise, we dynamically parse, generate code to a memory stream and then load the new assembly.

One of the issues with dynamically loaded assemblies is that references to them by other dynamically loaded assemblies are not automatically resolved in the way they are for assemblies that originate from disk. We therefore maintain a list of dynamically loaded assemblies and manually resolve to them by providing a custom handler for the `AssemblyResolve` event of our current application domain. One disadvantage of our dynamic generation and loading of assemblies is that they are never garbage collected (as occurs with the CLR light-weight code generation API used by IronPython [2]).

## 2.20 Command Line Arguments

Ruby has its origins on UNIX systems which support a rich range of command line globbing functionality. By the time the command line arguments find their way to the `Main` method of a CLR executable some information has already been lost (for example, the name of the executable and various parameter quote characters). So, rather than relying on the `args` array provided to the `Main` function, we instead call:

```
System.Environment.GetCommandLineArgs()
```

and manually perform our own globbing (analogous to how the win32 version of the C-Ruby interpreter works).

## 2.21 Symbols and Interning

The C-Ruby interpreter “interns” all strings used as class names, method names, etc, in a string table and then subsequently represents them by their integer offset within this table. We have so far, not performed such an interning and represent all of our methods names etc as CLR strings. This makes comparison slower but avoids the interning step. It would be nice if we could perform the interning at compile time, but this is not possible if we are to support separate compilation of Ruby source files as two symbols in different files with the same name should always be treated as identical. Interning of method names if one of many optimizations that we will soon experiment with.

## 3. OPTIMIZATION

Our focus so far has been exclusively on attaining the correct dynamic semantics of the Ruby language, and we have made no attempt to optimize the performance of the compiled code.

Nevertheless, we have attempted to avoid architectural features that would stand in the way of future efforts to improve the performance. Our aim is to incrementally add optimization steps in an environment where our regression test infrastructure can ensure that the changes do not affect correctness.

Interning method names as an optimization of dictionary lookup has already been mentioned in the previous section.

Mature implementations of dynamic languages invariably rely on call-site caching of method bindings to offset the overhead of repeated lookups. Although it is possible for such bindings to change, it is infrequent in practice. Thus caching is a big win. Such a mechanism is effective in the case of Ruby, and we expect

to implement the mechanism shortly. The challenge is to find the least conservative rule for invalidating bindings that still ensures correct behavior.

Finally, we are aware the computational data paths in the runtime libraries are far from optimal. As an example the main line of computation in the *equals* test for Ruby strings has a chain of method calls that between them perform no less than seven type instance tests before finally getting to start a character by character comparison. This can surely be improved, although it is a laborious task to ensure that all *non*-mainline control flow paths in the replacement code still provide correct semantics.

## 4. CONCLUSIONS

Our implementation of Ruby on the CLR has demonstrated that a verifiable code, compiled approach can achieve correct semantics. Our current implementation passes all tests for the functionality that we have implemented. As noted previously, the most significant missing functionality is threading and continuations. There is also some lack on functionality due to the fact that we have not implemented many of the libraries that are supplied as C-language interop libraries in the standard Ruby distribution. We expect that managed replacements for these will gradually be provided by us, or by others, as the need arises.

The “experiment” has highlighted once again the issues that arise in attempting new implementations of languages that are informally defined. In return we have been led rather more deeply

into the unique detailed semantics of the Ruby language. It has been a challenging and rewarding journey, and we hope that our experience will be of assistance to others who choose to tread the same path.

## 5. ACKNOWLEDGMENTS

We wish to thank Microsoft Research for their financial and technical support of this project.

## 6. REFERENCES

- [1] Gough, John.,” Compiling for the .NET Common Language Runtime (CLR)” Prentice Hall PTR, 2002
- [2] Hugunin, Jim. “Jim Hugunin’s Thinking Dynamic”, <http://blogs.msdn.com/hugunin/archive/2006/09/05/741605.aspx>
- [3] Ruby Language. <http://www.ruby-lang.org/en/>
- [4] Thomas, D, Fowler, C and Hunt, A. “Programming Ruby 2<sup>nd</sup> Edition” Pragmatic Bookshelf, 2004
- [5] Kelly, W. Gardens Point Parser Generator. <http://www.plas.fit.qut.edu.au/projects/LanguageProcessingTools.aspx>
- [6] Miller, J and Ragsdale, S. “The Common Language Infrastructure Annotated Standard” Addison-Wesley Professional, 2003