DEGREE PROJECT 2004:PM06

Division of Computer Science at the Department of Informatics and Mathematics

RubySharp – A Ruby to CIL Compiler

Jan-Åke Hedström

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

DEGREE PROJECT

University of Trollhättan · Uddevalla
Department of Informatics and Mathematics

Degree project for master degree in Software engineering

RubySharp – A Ruby to CIL Compiler

Jan-Åke Hedström

Examiner:
Dr. Robert Feldt Department of Computer Science

Supervisor:
Richard Torkar Department of Computer Science

Trollhättan, 2004
2004:PM06

EXAMENSARBETE

RubySharp – A Ruby to CIL Compiler

Jan-Åke Hedström

Sammanfattning

Det här examensarbetet beskriver utvecklingen av kompilatorn RubySharp. Det är en prototyp som kan kompilera delar av det dynamiska programspråket Ruby till CIL (Common Intermediate Language), vilket är ett statiskt assemblerspråk för .NET plattformen. Syftet är att kunna exekvera kompilerade Ruby program på .NET's CLR (Common Language Runtime).

Ruby och CIL jämförs för att belysa de likheter och skillnader som finns mellan programspråken och en RubySharp specifik klass hierarki utvecklas för att kunna kompilera in ett korrekt Ruby beteende i alla program. Vidare så definieras enkla målprogram i Ruby vilka översätts till CIL. Översättningarna ligger sedan till grund för hur RubySharp kommer att kompilera målprogrammen.

Även andra koncept i Ruby, förutom de som är definierade i målprogrammen, kommer att undersökas för att se om och hur de kan implementeras för att exekveras på .NET plattformen.

Utgivare:	Högskolan Trollhättan Uddevalla, Institutionen för Informatik och Matematik Box 957, 461 29 Trollhättan Tel: 0520-47 50 00 Fax: 0520-47 50 99		
Examinator:	Dr. Robert Feldt		
Handledare:	Richard Torkar, HTU		
Huvudämne:	Programvaruteknik	**Språk**:	Engelska
Nivå:	Fördjupningsnivå 2	**Poäng**:	10
Rapportnr:	2004:PM06	**Datum**:	2004-05-26
Nyckelord:	RubySharp, Ruby, .NET, Compiler		

RubySharp – A Ruby to CIL Compiler

Jan-Åke Hedström
Department of Computer Science
University of Trollhättan/Uddevalla
tds00jahe@thn.htu.se

Abstract

This paper describes the development of RubySharp, a proof-of-concept compiler. It is able to compile a subset of features from the dynamic programming language Ruby into CIL (Common Intermediate Language), which is a static assembler intermediate language in the .NET platform. The language constructs of Ruby and CIL are examined to see what similarities and differences that exist. To be able to support the correct Ruby behaviour in all compiled programs, a class hierarchy is developed for RubySharp. Additionally, basic goal programs are defined in Ruby and in CIL, forming a specification for the RubySharp compiler on how to compile the goal programs. Furthermore, other Ruby language constructs, besides these introduced by the goal programs, are also examined to see if and how they can be implemented to the .NET platform.

1. Introduction

Since the release of the .NET platform in the year 2000, .NET has grown to become a major part of the development community. The .NET platform integrates several technologies developed by Microsoft in the 1990s, such as programming languages, development tools, enterprise servers and the .NET framework.

The .NET framework is a complex architecture, with design goals such as language integration, application interoperation over networks, development and deployment. However, one of the most important and interesting design choices for the .NET framework, at least from this project's point-of-view, is the CLR (Common Language Runtime) [1]. The CLR defines a runtime engine which among other tasks loads required classes, performs just-in-time compilation and security checks on executed code. It is also designed as a platform targeted for several programming languages.

The CLR infrastructure supports all languages that can be represented in CIL (Microsoft's Common Intermediate Language) [1]. These languages go by the name .NET languages and currently, there are four Microsoft supported languages: C#, Visual Basic.NET, Managed C++ and JScript.NET [1]. To add the ability for the .NET languages to interoperate, the CTS (Common Type System) [1] has been developed.

The CTS is static and explicit. It means that types executed under the CLR are known at compile-time and the types must be explicitly stated when declared. Due to Microsoft's extensive development of the .NET platform, a large part of the development community has taken interest in .NET. Research is targeted towards .NET, and many third-party programming languages have been implemented and incorporated into the .NET platform. Examples of other programming languages also being static are Eiffel#, Haskell, FORTRAN, and COBOL [1].

The CLR and the CTS were not designed with dynamic programming languages in mind. Therefore it is interesting to see that some of the third-party programming languages implemented for .NET are dynamic. Perl, Python and Smalltalk are examples of programming languages which type systems are dynamic and implicit. In a dynamic language an object's type is determined during runtime, and implicitly states that a type does not have to be specified when declared. When implementing third-party programming languages for the .NET platform, the developers must either extend the functionality of the CIL to get the dynamic behaviour for the implemented language, or decrease the flexibility of a certain language construct in the implementation.

The rather large interest in integrating other programming languages for the .NET platform is mainly because of three reasons. First, the extensive investment done by Microsoft vouches for further development and support of .NET. Second, there exist

a lot of source code and documentation for the .NET platform and for various kinds of applications making it easy for developers to find information. Third and finally, it offers integration to an extensive development platform so that non-.NET languages can be used in applications developed for the .NET platform.

Currently, no compiler exists able to compile the dynamic programming language Ruby for execution on the .NET platform. For this purpose, a proof-of-concept compiler, RubySharp, will be developed and a comparison of the Ruby and .NET environments will show important similarities and differences in language constructs.

1.1 Related Work

Some example implementations of dynamic languages to .NET are #Smalltalk (Sharp Smalltalk) [2], IronPython [3, 4], Python for .NET [5] and Perl for .NET [5, 6]. The #Smalltalk compiler is the implementation with the most available information, and its design and functionality is described in more detail in Section 3.

IronPython [3, 4] implements the Python programming language for the .NET platform. It compiles Python programs so that they will run natively on the CLR. The compiler includes both an interactive interpreter like standard Python and a static compiler, compiling Python source code into .NET binaries (.exe or .dll). In addition, IronPython can use CLR libraries and also has the ability to extend .NET classes.

The main focus of IronPython has been performance, making a fast implementation of Python. The performance measurements presented at [3] establish that dynamic languages can have good performance when executed on the CLR. In [3] it can also be read "Allowing a dynamic language to run well on this platform requires careful performance tuning and judicious use of underlying CLR constructs". There are also rules specified which provides guidelines for attaining performance goals [3]. First, use native CLR constructs as often as possible because they are optimized by the runtime engine. Second, use dynamic code generation to shorten time for common cases, e.g. using *System.Reflection.Emit* [7] to produce CIL code on the fly. Third, to have fallbacks either for cases not so common, or when a fully dynamic implementation is required. The fourth and final rule is to always test performance on language constructs that are used and to find other solutions when performance is too poor. As an example, Type.InvokeMember should never be used if good performance is required [3]. IronPython is currently a research prototype that has not yet been released. No further information about IronPython's design and implementation has been found and there is no source code available at this time.

Another Python implementation for the .NET platform is the Python for .NET compiler [5]. The compiler supports cross-language inheritance between Python and .NET and uses the *System.Reflection.Emit* [7] API (Application Programming Interface) for compilation. During the development, existing code from e.g. the CPython parser and the Python infrastructure were used. Much of the code is written in Python with the drawback of AST (Abstract Syntax Tree) manipulation being slow. The Python for .NET runtime is written in C#, and the runtime defines an interface that captures Python's dynamic semantics. In contrast to IronPython's performance, the performance of Python for .NET is considered to be poor, making it useful only for demonstration purposes.

Perl for .NET [5, 6] is a research project with the aim to implement a native code compiler for the .NET platform. The status for the Perl for .NET compiler is experimental, supporting only a small part of the Perl programming language. The research project started in the year 2000, when Microsoft's *System.Reflection.Emit* [7] API still was under development, leading to frequent changes in the API. Therefore, the Perl for .NET compiler used C# as the intermediate language instead. Even though Perl for .NET is limited, it supports instantiation of .NET objects, access to .NET methods and properties, implementation of .NET classes, and cross-language inheritance between both Perl and .NET classes. During the development of Perl for .NET, the problems that existed were mainly concerned with the execution speed and full language support. The execution speed were approximately ten times slower than the Perl interpreter, and one reason is that in .NET, objects had to be checked for correct type. Finally, Perl for .NET does not have an implementation of *eval* (explained in Section 2.1) because of the runtime support required. *Eval* requires re-entering the parser and the code generator at runtime which were difficult because of Perl for .NET's design.

1.2 Goals

The goal for this project is to develop a proof-of-concept compiler, RubySharp, and to explore the possibilities of compiling Ruby source code into CIL for native execution on the CLR. To achieve this goal it is necessary to understand the differences between the dynamic language Ruby and the static .NET environment and to examine the runtime support needed to implement correct Ruby behaviour into CIL. Basic goal programs will be defined in Ruby, which

will be translated into CIL. The goal programs will include common features such as e.g. printing strings, defining classes, calling methods and inheritance. They will also be defined in increasing order of difficulty, each introducing new Ruby constructs. The CIL translations will function as the foundation on how the RubySharp compiler will compile the goal programs. Also, understanding the design of the #Smalltalk compiler describing its advantages and disadvantages can provide useful information when making design choices for RubySharp.

Limitations that will not be considered during the development of RubySharp are first, performance issues. The initial goal is to get a functional proof-of-concept compiler showing the feasibility of RubySharp. Therefore, the performance at this stage of the development is of minor significance.

Secondly, support of all Ruby's basic classes. The first version of RubySharp will contain only a subset of Ruby's classes, beginning with *Object* and *String*. More advanced language constructs such as IO, Threads and file handling is saved for later versions.

Third and finally, not all parts of these classes will be implemented, only the parts that is needed for compiling the goal programs.

The RubySharp compiler will be developed and released under the open-source license GPL (GNU Public License) [8], so that the result can be free of use for other developers.

In the following section, Section 2, the dynamic programming language Ruby and the static .NET platform is examined. Section 3 presents the functionality of the #Smalltalk compiler and what has, and has not been implemented from the Smalltalk programming language. In Section 4, how Ruby concepts are transferred into CIL is described together with details on the goal programs and RubySharp's compilation process. Section 5 contains discussions on RubySharp implementation followed by future work in Section 6 and Conclusions in Section 7.

2. Ruby and .NET

There exist a lot of differences between a dynamic language such as Ruby and the static environment provided by the .NET programming languages. This section examines how the different environments work and which language constructs exists.

2.1 Ruby

The current version of Ruby is 1.8.1 installer version 12, released on April 18, 2004. The following Ruby information in this section is based on *Programming Ruby – The Pragmatic Programmers Guide* [9] and *An Invitation to Ruby* [10].

Ruby is a fully object-oriented programming language, i.e. everything is an object, even integers and numbers. It has a dynamic type system (as Python and Smalltalk). Furthermore, Ruby is an interpreted programming language with a garbage collector and its runtime is available for most common operating systems. In Figure 1, a small part of the Ruby class hierarchy is shown.

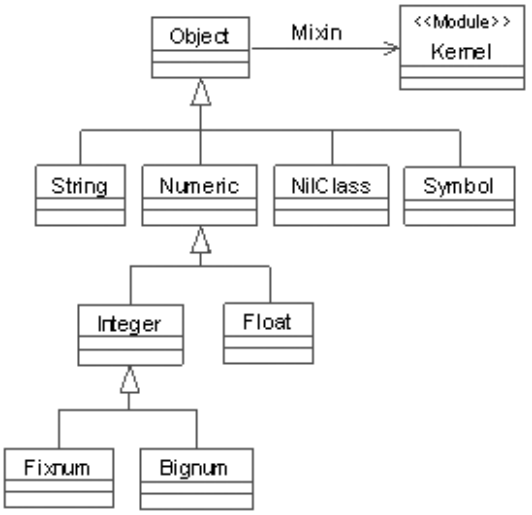

Figure 1. Ruby class hierarchy.

At the top level of the class hierarchy is *Object*, the class from which all Ruby objects derive. Figure 1 also shows the inheritance structure of classes and modules that are relevant for the current RubySharp development. *Fixnum*, *Bignum*, *Symbol* and *Kernel* are described further down in this Section. *String*, *NilClass* and *Kernel* are discussed in Section 4.1.

In Ruby, no entrypoint method exists such as e.g. C#'s *public static void Main()*. The executing source code is placed outside the scope of any user-defined class and is executed automatically by the Ruby interpreter.

Ruby has the same language constructs as most object-oriented programming languages, such as classes, methods and variables with a few conceptual differences. The next paragraphs will describe how these concepts are used in Ruby.

Ruby is a single-inheritance programming language and classes are defined between *class/end* keywords. All classes in Ruby are open which in this case means that all classes are by default public and cannot be restricted with e.g. the private or protected access modifier. Classes can be re-opened anywhere in the

code to add variables and methods. It is even possible to redefine previously defined methods and there are no limitations on which classes and how many times redefinition can occur. Listing 1 clarifies this concept.

```
class A
  def method()
    puts “old”
  end
end

a = A.new
a.method            // output: “old”

class A
  def method()
    puts “new”
  end
end

a.method            // output: “new”
```

Listing 1. Open classes.

In methods, Ruby allows for three levels of protection which is provided through the access modifiers *public*, *private* and *protected*. All methods are by default public, except *initialize* (Ruby's constructor) which is private. Overloading in Ruby is not allowed, i.e. multiple methods cannot have the same name and differ in number or types of arguments. If a method is defined using an existing name, the method is redefined making the previously defined method unavailable. Furthermore, explicit return statements are not needed in Ruby and at the same time, it is not necessary to specify return types when defining methods. The value of the last executed statement is always returned to the caller. Two kinds of methods exist (Listing 2), *instance methods* (defined using one of the three access modifiers and available on instantiated objects) and *class methods* (available without the need of instantiation).

```
class Example
  // instance method
  def method1
  end
  // class method
  def Example.method2
  end
end
```

Listing 2. Instance and class methods.

Three types of variables exist in Ruby: *local*, *instance* and *class* variables. Local variables are only available within the scope they are used, e.g. within a method. Instance variables always begin with '@' and are available within the instance of the class. Finally, class variables begin with '@@' and are shared by all instances of the class.

Besides the common concepts described, Ruby offers other features, some more common in dynamic programming languages than in static. These are e.g. *iterators*, *blocks*, *mixins*, *method_missing*, *introspection* and *eval*, which is described in the forthcoming paragraphs.

Iterators are used to loop through elements in an array or through a collection. It is common to use blocks together with iterators and there exist a number of ways to iterate in Ruby. In Listing 3, three examples of iteration are shown.

```
3.times {print “*”}
1.upto(3) {|n| print n}
(‘a’..’c’) {|char| print char}

// Output: ***123abc
```

Listing 3. Iterators.

A block is a group of statements defined within curly braces or between *do/end* keywords. Blocks can be used to pass source code as arguments to methods and are often used together with iterators (previously shown in Listing 3). A method that takes a block as a parameter can execute that block once or more by calling *yield*. Within a block it is possible to access variables existing in the block's current scope as well as to define local block variables which are declared within vertical bars, '|'. An example of a block is presented in Listing 4 where an array is created, iterated through with *each* and printed.

```
arr = “dog”, “cat”, “horse”
arr.each {|animal| puts animal}

// Output:  dog
//          cat
//          horse
```

Listing 4. Blocks.

Furthermore, it is also possible to return values from a block. Consider the code in Listing 5, where *arr* uses the iterator *find*, and for each element in *arr* checks to see if it equals “Hello”. If it is equal it is returned and assigned to *var*.

```
var = arr.find {|word| word == “Hello”}
```

Listing 5. Block returns.

In *Object*, a method named *method_missing* exists. It handles method calls to an instance on which the invoked method do not exist. By default,

method_missing raises an exception, printing an error message. When overridden, *method_missing* can effectively be used in programs to relay method calls such as in the example in Listing 6. It can also be used for overloading methods since differences in method signatures are not allowed in Ruby. In these cases, *method_missing* is overridden defining how the arguments to the method call should be handled.

```
class Roman
  def romanToInt(str)
    # Conversion from roman to integer
  end
  def method_missing(methId)
    str = methId.id2name
    romanToInt(str)
  end
end

r = Roman.new
r.iv        // Output: 4
r.xxiii     // Output: 23
```

Listing 6. method_missing.

In Listing 6, a class is defined which converts roman numbers to integers. By using the notation *r.iv*, *method_missing* is automatically called sending in *iv* as a parameter. The parameter accepted by *method_missing* is of type *Symbol* and is converted to *String* by *Symbol's* instance method *id2name* and sent to the method *romanToInt.*

Ruby is a single-inheritance programming language but by using *mixins*, similar functionality existing in multiple inheritance languages can be included. Mixins are done with modules which can be used in two ways, both as a namespace and as a mixin. An example of modules used as a mixin is the *Kernel* module that is mixed-in to the *Object* class (Figure 1). Modules cannot be instantiated so methods defined within the module are called module methods (similar to class methods). But when modules are used as a mixin, a class can include a module by using the include keyword. The methods within the module are added to the class as if they were already defined within it. Also, modules are included by reference, i.e. when multiple classes include the same module, the classes point to the same module. This is different from module instance variables that are included per object.

Mixins behave in a similar way as super classes, making the methods available for the including class and its derived subclasses. An example of a powerful Ruby mixin is *Comparable* which have the possibility to interact with the code within the including class. The comparable operator "<=>" must be defined giving the possibility to use other comparable operators (<, <=, ==, >=, >).

Introspection (called reflection in other languages) is the ability to examine parts of a program from within the program itself. When using introspection in Ruby it is possible to discover which objects that exist, the current class hierarchy, an object's contents and behaviour and finally method information.

Another language construct in Ruby is *eval* which makes it possible to parse and execute Ruby source code dynamically from within the program. This source code can be defined in strings or blocks or even imported from a web page form. In Listing 7, an example of *eval* is shown. The example shows a string that is dynamically retrieved during program execution from a cgi script and stored in the variable *expr*. The string is executed within the *begin/rescue* statement handling possible exceptions.

```
expr = cgi["expression"].to_s

begin
  result = eval(expr)
rescue Exception => detail
  # handle bad expressions
end
```

Listing 7. Eval.

Two additional versions of *eval* exist in Ruby: *instance_eval* (executes source code within the receiver giving access to its instance variables and methods) and *module_eval* (adds methods to a class and also goes by the name *class_eval*).

Finally, some dynamic programming languages, such as e.g. Ruby and Smalltalk, handle integers in a different way than most statically typed programming languages. Ruby has the classes Fixnum and Bignum that derive from Integer. Fixnum is an immediate type, i.e. they are passed by value when sent as an argument. Integers in Ruby cannot overflow, i.e. when adding 1 to the maximum Fixnum number it is converted to a Bignum. Bignum has infinite precision but is not immediate, i.e. it is passed by reference. Whenever a Bignum operation results in an Integer that could fit in a Fixnum, it is converted back.

2.2 .NET and CIL

The current version of the .NET framework is 1.1.4322. The .NET framework is available for the Windows operating system but is currently implemented for the Linux/Unix platform in the Mono [11] project. The following .NET information is based on *.NET Framework Essentials* [1], *Inside Microsoft .NET IL Assembler* [12], *CIL Programming: Under the Hood of .NET* [13] and *Programming C#* [7].

The .NET programming environment offer developers the opportunity to develop applications in several independent programming languages. A characteristic about .NET is the multiple language integration which is possible because of the CTS specification all .NET languages adhere to. It means that e.g. inheritance, exception handling and polymorphism can take place between objects in different programming languages. When the .NET languages are compiled, they are compiled into CIL that is executed by the CLR.

The Microsoft supported .NET languages are object-oriented. They are also static and explicit. Furthermore, .NET only allows for single-inheritance, but classes can implement multiple interfaces. In addition, the .NET runtime is a garbage collected environment.

In Figure 2, an extract of .NET's class hierarchy is shown.

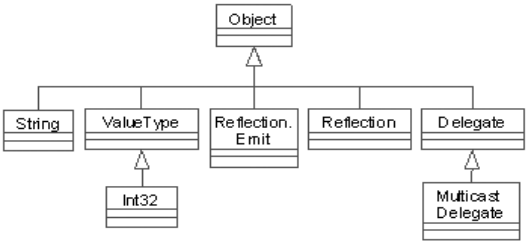

Figure 2. .NET class hierarchy.

In .NET, *Object* is at the top level and it is the class from which all other classes derive. The extract of *Object's* derived classes in Figure 2, is relevant for the current RubySharp implementation. *String* and *Int32* [1] are discussed further down in this section when value types and reference types are discussed. *System.Reflection* [7], *System.Reflection.Emit* [7] and *System.Delegate* [7] are discussed both in this section and in Section 4.1.

For the CLR to be able to load a .NET executable (.exe or .dll), the binary file must conform to a specific file format, the PE (Portable Executable) file format. It is a derivative of Microsoft's COFF (Common Object File Format) file format and is responsible for storing static assemblies to file. In .NET, basically two types of assemblies exist: *static* and *dynamic*, where static assemblies are stored in files. Dynamic assemblies are dynamically created at runtime by the *System.Reflection.Emit* [7] API and are stored formatted in the PE file format in memory.

An assembly is a collection of metadata and CIL source code. Metadata is information about assembly members such as namespaces, classes, methods and fields and also about the assembly itself. A manifest is always present in an assembly as part of its metadata. The manifest holds information about the assembly contents such as identification information (version and name), a list of all types in the assembly and a map to connect them to the CIL source code and finally a list of all assemblies referenced from within the current assembly. The assembly and module information forms the program header in CIL and an example is shown in Listing 8.

```
.assembly extern mscorlib{}
.assembly MyAssembly{}
.module MyAssembly.exe
```

Listing 8. Assembly information.

The first line is the metadata item *Assembly Reference* which identifies the external assembly, in this case the main assembly *mscorlib.dll* [6]. It defines all base classes from which other classes derive. The *Assembly* metadata item on line two identifies the current assembly. It identifies the application and is executed when the application starts. The *Assembly* is always named with its file name without the extension. The last line defines the *Module* metadata item identifying the current module by its file name including the extension. It is possible for multiple modules to exist in the same assembly.

For RubySharp, it is important to know what language constructs CIL can support. Since the .NET languages are compiled into CIL, CIL must support the differences in all .NET languages. Each of the .NET languages has restrictions which are upheld by each compiler, e.g. the CSC and VBC compilers for C# and Visual Basic.NET. As an example, CIL allows usage of characters when naming objects that C# does not, e.g. the '$' character. Also, CIL allows for usage of global variables and methods, that e.g. C# do not. This leads to a flexible intermediate language allowing for several programming language constructs.

CIL is a stack-based programming language. When using objects they must first be loaded onto the stack. And in the cases where operations return values, they are placed on the stack to e.g. be stored in variables.

In CIL, namespaces are declared using the *.namespace* keyword. Adding namespaces avoids naming conflicts when reusing source code from other applications. In Listing 9, an example of a namespace declaration in CIL is shown.

```
.namespace RubySharp
{
  ...
}
```

Listing 9. Namespace declaration.

Classes are declared in a *Type Definition* metadata item using the keyword *.class*. Classes can use the access modifiers *public* or *private* (default). Other keywords are also used in the declaration. The layout style flag (*auto*, *sequential* or *explicit*) tells the loader how to layout the class in memory. The string conversion flag defines how strings are handled when operating with unmanaged code. The default is *ansi* (normal C style string conversion) but *unicode* and *autochar* (using the underlying platform's convention) exists as well. Another flag is the *beforefieldinit* flag, which specifies that the runtime will handle type initialization before any methods are invoked on the instance. Furthermore, an *extends* keyword define the parent of the class. If no user-defined parent exist *[mscorlib]System.Object* is always used. All classes derive from the *System.Object* [1, 7] namespace which is defined in the *mscorlib.dll* assembly. If a class implements any interfaces they are added to the class declaration using the *implements* keyword directly after the super class definition. A typical class declaration is shown in Listing 10.

```
  .class auto ansi beforefieldinit MyClass
extends [mscorlib]System.Object
  {
    ...
  }
```

Listing 10. Class declaration.

When declaring classes in CIL, a technique called *class amendment* can be used. It means that a class declaration can be split over multiple locations in the source code, i.e. a class can be reopened to add e.g. methods and fields. All characteristics of a class should be declared in the first definition, all keywords used (flags, *extends*, *implements*) in amended classes are ignored.

Methods are defined in the metadata item *Method Definition* with the flags *access modifier*, *type modifier* and *return type*. Allowed access modifier flags for CIL are besides the common, *public*, *private* and *family* (similar to protected), also *assembly* (accessed from anywhere in the assembly), *famandassem* (accessed from any derived class within the assembly), *famorassem* (accessed from anywhere inside the assembly and to derived classes outside of the assembly) and *privatescope* (default). The type modifier can be either *instance*, *virtual* or *static*. If *instance* is specified, the method belongs to an actual instance of a class. *Virtual* defines that a method can be overridden in a derived class hiding the virtual method in the super class. When overriding a method, the methods in derived classes are also declared as virtual. They will overwrite the overridden method unless the virtual method in the super class is declared with the *newslot* keyword. In that case, the method is added to a new slot in the *virtual method table* for that class. The third type modifier is *static*, which implies that the method is a class method available for all instances of the class. The return type defines what type of object is left on the stack after the method has been executed. In Listing 11 a method declaration is shown.

```
  .method public virtual newslot void
MyMethod() cil managed
  {
    .entrypoint
    .maxstack 1
    ...
  }
```

Listing 11. Method declaration.

Cil managed are implementation flags defining that the method is implemented in CIL and managed by the runtime. If the method was represented in native code the flag would be *native unmanaged*. Furthermore, in C# *public static void Main()* is used as the entrypoint for the application. In each assembly only one entrypoint can exist which is defined with the *.entrypoint* keyword. Also, there are no naming restrictions on the entrypoint method in CIL. The one restriction is that it must be static. Finally, since CIL is a stack-based language, every time a method is entered a *.maxstack* value defines how many slots on the stack the method uses. If not specified, the default value is 8.

Two other types of methods are instance constructors and class constructors. Instance constructors (which are analogous to C++ constructors) are specific to an object's instance and initialize an objects instance fields. Instance constructors have the name *.ctor* and can have parameters but must return void. They must be declared with the keywords *instance*, *specialname* and *rtspecialname*. The last two are for internal use by the runtime only. Instance constructors are usually called with the *newobj* instruction but can also be called explicitly, resetting the state of the instance.

Class constructors (also called type initializers) are specific to an object as a whole. It is loaded before any of the objects members are accessed, but after the object itself is loaded (they are not called from within the code, but from the runtime itself). Class constructors are used for static field initialization. They

do not have any parameters and always return void. They are declared with the name *.cctor*, the flags *specialname* and *rtspecialname* and they are always *static*. Class constructors can be called explicitly, resetting the static values. In Listing 12, the two constructors are shown.

```
  // Default instance constructor
  .method public specialname rtspecialname
  instance void .ctor() cil managed
  {
    .maxstack 1
    ldarg.0
    call instance void
[mscorlib]System.Object::.ctor()
    ret
  }

  // Class constructor
  .method private specialname rtspecialname
static void .cctor() cil managed
  {
    ...
  }
```

Listing 12. Instance and class constructors.

Two different types of variables are used in CIL: fields and local variables. Fields are declared using the keyword *.field* and they are defined in the metadata item *Field Definition*. Fields as well as methods, if declared within a class scope, belongs to that class, but when declared outside the scope of a class they are global. The same visibility flags and type modifiers are used for fields as for methods. Local variables are only visible within the scope of a method and when defined with the keyword *init*, the variable must be initialized before the method executes, to avoid errors. In Listing 13, examples of *.field* and *.locals* declaration are shown.

```
  .field public static int32 VarName
  .method public static void Main()
  {
    .locals init (int32 VarName)
    ...
  }
```

Listing 13. Field and local variable declaration.

Other .NET and CIL constructs described here are *interfaces*, *delegates*, the *System.Reflection.Emit* [7] API and finally, *value* and *reference* types.

Interfaces in .NET are used as contracts. They define how a class or a struct will behave. When implementing an interface, all methods, properties and events defined in the interface must be implemented within the implementing class. Limitations of interfaces are that they cannot derive from classes or other interfaces and they cannot have instance methods. .NET classes can however implement multiple interfaces extending the single-inheritance functionality by adding capabilities to a class. In Listing 14, an example of an interface declaration is shown.

```
  .class interface public abstract auto ansi
IMyInterface
  {
     .method public newslot abstract virtual
instance void MyMethod() cil managed
    {
    }
  }
```

Listing 14. Interface declaration.

Delegates [7] are reference objects and function pointers used to encapsulate methods having a specific signature and return type. They are always implemented as *sealed*, denying any other class to derive from them. Furthermore, delegates are implemented in CIL as nested classes with two mandatory methods, a constructor and *Invoke* when used on the same thread as the application. If a separate thread is used for the delegate, two additional mandatory methods must be defined: *BeginInvoke* (starts thread) and *EndInvoke* (ends thread). All of these four methods are declared as *virtual* and defined with the *runtime managed* flag, i.e. they do not have a CIL implementation and are completely managed by the runtime. Delegates are often used together with events in .NET. The CIL code for delegates is extensive and hard to understand. Therefore, an example of a delegate usage is provided in C# (Listing 15).

```
public delegate void Del();
public Del DelName = new Del(method)

public void method()
{
  Console.WriteLine("Delegate");
}

DelName();          // Output: Delegate
```

Listing 15. Delegates in C#.

The *System.Reflection* [7] and *System.Reflection.Emit* [7] namespaces are used to access, examine and interact with metadata. Reflection is mainly used for three tasks: viewing and displaying metadata (such as e.g. classes, methods and fields), examining assembly objects for interaction or instantiation and finally, for *late binding*, which allows for invoking methods and properties on dynamically

instantiated objects. *System.Reflection.Emit* [7] is used to create new objects during runtime to perform specific tasks. This is useful when e.g. a class dynamically created at runtime will run much faster than when created at compile-time. The *System.Reflection.Emit* [7] API can also create static assemblies, i.e. store them to binary files (.exe or .dll).

Finally, .NET supports two different types of objects: *value* types representing an actual value stored on the stack and *reference* types, storing only a reference to an object on the heap. Value types cannot be *null* and when sent as parameters they are sent by value, i.e. a copy of the original is sent. Examples of value types are *int*, *struct* and *enum*. Reference types, when used as parameters, saves resources since they are not sent as copies but as pointers to the address of the actual object. Examples of reference types are *class* instances, *interface*, *string*, *array* and *delegate*.

3. #Smalltalk Compiler

The following information on the design and implementation of #Smalltalk has been taken from the #Smalltalk (Sharp Smalltalk) web site [2] and a power-point presentation [14].

3.1 Overview

#Smalltalk is a standalone compiler, able to compile Smalltalk source code written in the ANSI (American National Standards Institute) SIF (Smalltalk Interchange Format) format into binary files to run natively on the .NET framework.

The compiler uses a base class library which is compatible with the ANSI Smalltalk Standard. #Smalltalk also allows usage of .NET classes from e.g. the *mscorlib.dll* [6] library. Furthermore, .NET classes and methods can be referenced from the Smalltalk source code. This is done by using the full name, e.g. *System.Object* [1, 7]. When referencing e.g. external methods the compiler handles the conversion of objects between #Smalltalk objects and .NET objects. An example of such conversion is the handling of integers. Smalltalk uses the types SmallInteger and LargeInteger. SmallInteger has a fixed byte size of four bytes and LargeInteger has a variable byte size handling any integer precision. .NET integer types on the other hand, only have fixed sized integers, e.g. *int16*, *int32*, *int64*, *uint16*, *uint32*, *uint64* [1]. For larger integers, the *decimal* [1] type can be used while also having limitations in range.

As in Ruby, Smalltalk integers cannot overflow. If adding one to the largest possible SmallInteger, a LargeInteger is returned. This conversion is normally handled by the Smalltalk runtime, but to provide the same functionality in Smalltalk programs compiled for the .NET platform, the #Smalltalk developers provide the class LargeInteger in the file LargeInteger.dll. Furthermore, there might be overloaded functions in .NET that differs in parameter types such as *int32* or *int16* [1] parameters. When that occurs, the #Smalltalk compiler just chooses a function, sometimes leading to wrong choices. To avoid this, the developer must convert the #Smalltalk integer to another #Smalltalk class implementing the correct integer type.

3.2 Implementation

In the implementation of #Smalltalk, all classes inherit from a #Smalltalk *Root* class which is derived from .NET's *System.Object* [1, 7] class. By implementing the *Root* class, the problem of static typing in .NET is avoided. All objects used in a #Smalltalk compiled program such as instances, arguments and return types will be created as *Root* to mimic the dynamic type system in Smalltalk. Furthermore, the *Root* class implements the *#doesNotUnderstand* method which is similar to Ruby's *method_missing*. In Figure 3, the #Smalltalk class hierarchy is shown.

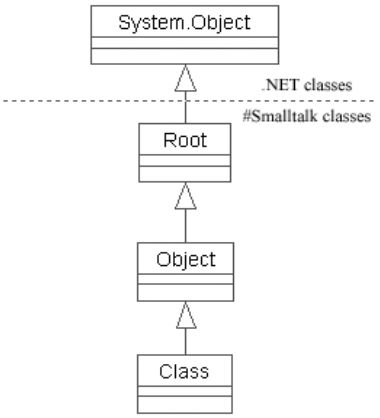

Figure 3. #Smalltalk class hierarchy.

For a correct implementation of Smalltalk semantics in all compiled programs, the #Smalltalk developers have built the #Smalltalk class hierarchy. It starts with the top level *Object* which is subclassed under the *Root* class. All Smalltalk's classes and meta classes are directly represented by .NET classes. The Smalltalk classes which are used in a program will also have an implementation in the #Smalltalk hierarchy. All classes that are user-defined in a Smalltalk program will

inherit *Object* in the #Smalltalk class hierarchy as they would inherit from *Object* in Smalltalk.

To avoid the problem of static type checking all classes that are instantiated, both #Smalltalk classes (e.g. *String* and *SmallInteger*) and user-defined classes will have the type *Root*. This means that all arguments and return types also are *Root*. To execute the correct methods on an instance, all methods in #Smalltalk classes are declared *virtual*.

There are three types of variables used in Smalltalk: *instance*, *class instance* and *class* variables. The instance variables and class instance variables are implemented as instance variables in .NET, the later on the .NET class corresponding to the meta class in Smalltalk. Class variables in Smalltalk are represented by static variables in .NET.

All blocks in Smalltalk are compiled into classes in .NET. All references to variables within the block that are not defined are created as instance variables of the class. When the block (read: class) is instantiated in .NET all external variable references are passed in as arguments.

More difficult are blocks returning values. Returns that are non-local, i.e. returning objects outside of a method's scope are not supported in .NET. To solve this, #Smalltalk uses exceptions. Whenever a method is entered returning a non-local value, a number is generated. The thrown exception (with the generated number), is handled by the throwing method returning the value of the exception. The return exception is returned non-locally if the generated number does not match the exception's return.

3.3 Future Work of #Smalltalk

Although #Smalltalk supports a major part of the Smalltalk programming language, there are still items which need to be implemented and further developed for a complete Smalltalk for .NET implementation. The following are examples specified by the #Smalltalk developers from their web site [14].

#Smalltalk does not have the ability to subclass .NET classes. Currently, all created objects must inherit from the *Root* class. If a .NET class was subclassed, these classes would not have the same type as the other #Smalltalk objects. To make inheritance from .NET classes possible, a *Root* interface can be used instead which defines all necessary methods to be implemented. It could also be implemented by creating a subclass for the .NET object and a #Smalltalk class wrapping the .NET subclass. Two classes will in this case be created for every .NET subclass but should be transparent to the developer.

Other future implementations for #Smalltalk is to add the possibility to name #Smalltalk types and method names so that they can be exported to the .NET environment using the same name. Furthermore, the #Smalltalk class library could be developed further, adding support for more than the Smalltalk ANSI standard classes such as databases and GUIs. .NET's classes could be used for this purpose but might need to be wrapped.

4. RubySharp Compiler

This section describes the actions taken during the development of RubySharp. First, a description on how the Ruby behaviour is transferred into CIL is made followed by a detailed description on the goal programs. Finally, the RubySharp compiler is described in detail.

4.1 Combining Concepts

In Section 2, the language constructs of Ruby and CIL were explained. In this section the concepts of these two programming languages is combined so that the correct Ruby behaviour can be implemented into CIL. Not all language constructs described in Section 2 have been implemented in RubySharp and if not discussed in this section they are mentioned in forthcoming sections.

To be able to understand how Ruby concepts are best implemented in CIL, small programs called *spikes* have been used. A *spike* is a well delimited program testing only small features. Spikes have been used, both in Ruby, C# and CIL to examine different behaviour in language construct and the result is presented in the remaining part of this section.

All classes in Ruby are always accessible and open. To accomplish this in CIL, all classes are implemented with the *public* keyword making it visible for all objects. Since classes in CIL can be declared partially (*class amendment*), it also mimics Ruby's open classes such as adding contents to a class, even though it is limited in the aspect of redefining methods within a class. The flags that are used when declaring CIL classes are besides public also *auto*, *ansi* and *beforefieldinit*. Also, inheritance involves no problems since both Ruby and CIL are single-inheritance languages.

The constructors in Ruby are private by default and public in CIL. However, the semantics on how classes are instantiated is similar and straightforward to implement.

Ruby instance methods have three access modifiers, *public*, *private* and *protected*, which can be directly translated into CIL's access modifiers *public*, *private*

and *family*. All instance methods are declared as *virtual* and class methods are *static* in CIL.

The Ruby local variable type is translated into CIL's *.locals* and the instance and class variables is both translated to CIL's *.fields*, either declared with the keywords *instance* or *static*.

Modules can be used in two ways in Ruby, either as a namespace or as a mixin. They are differently implemented in CIL, and when used as a namespace modules can be directly translated into a CIL namespaces. For mixins, two solutions are available. First, the module can be implemented as a class and instantiated into all including classes as an instance variable. Whenever a method is invoked on the module, RubySharp uses the instance variable to access the method. Second, all the module's methods and variables can be copied into the included class. This solves the problem of the including classes getting the module's instance variables as their own instance variable. However, there will be an overhead with multiple copies of the same method if more than one class includes the same module.

However, when redefining methods several solutions exist. First, in the cases when only derived methods are overridden these can be declared *virtual* in the CIL subclasses as well. Second, for all cases where the first solution is not enough and all redefinitions are known at compile-time, delegates can be used explicitly redirecting the delegate to the correct method at a certain time during the execution. Third and finally, RubySharp can create and execute dynamic assemblies using the *MethodRental.SwapMethodBody* [15] within *System.Reflection.Emit* [7] to exchange the CIL code within a method. These solutions apply to *static* methods as well but when they are overridden the keyword *new* is used instead of *virtual*.

To be able to implement the correct Ruby behaviour when compiling programs for the .NET platform, a RubySharp class hierarchy has been developed. It has been developed in C# and it is implemented in such a way that, as close as possible, Ruby syntax can be used. After compiling the RubySharp class hierarchy, CIL source code is extracted using *ildasm.exe* (IL Disassembler) [12]. The complete extracted CIL code is then compiled into all programs compiled by RubySharp. In Figure 4, the implemented RubySharp class hierarchy is shown (Appendix A for complete source code in both C# and IL).

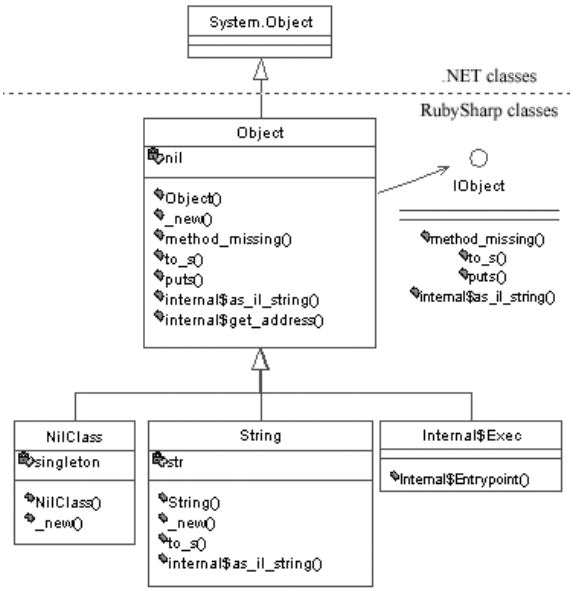

Figure 4. RubySharp class hierarchy.

RubySharp's class hierarchy inherits from .NET's *System.Object* [1, 7] class making it possible to provide compiled programs with the correct Ruby behaviour. The structure of the RubySharp class hierarchy corresponds to the class hierarchy in Ruby (Figure 1). The *Object* class represents Ruby's *Object* class including everything that is needed to mimic the correct Ruby behaviour. In the current version of RubySharp, *Object* includes *method_missing* and *to_s* from Ruby's *Object*. *Puts* is a class method that exist within the module *Kernel* which is mixed-in to *Object*. The *Kernel* methods are implemented as static methods in RubySharp's *Object*, mimicking correct Ruby semantics.

To instantiate objects within the RubySharp class hierarchy all classes define the static method *_new*. The *_new* method wraps the .NET constructor returning an instance of correct type when instantiated.

The RubySharp object *String* is a wrapper for the .NET class *System.String* [7]. Whenever a string is used in a Ruby program, it is an instance of *String* that is used. The actual .NET string is stored in the instance variable *str* and is accessible through the method *internal$as_il_string*.

Methods beginning with the name *internal$* are for use by RubySharp only and corresponding methods do not exist in Ruby's class hierarchy. *Internal$as_il_string* are used for retrieving the wrapped .NET string from a *String* object to e.g. print it to the console. The *internal$* methods are a part of the RubySharp class hierarchy and are mainly used for

accessing .NET types or for CIL specific purposes such as the *Internal$Exec* class. The *Internal$Exec* class inherits from *Object* and holds the entrypoint of the compiled program in the method *Internal$Entrypoint*. In this method, the executed source code from the Ruby program is placed. *Internal$* methods cannot be invoked from Ruby source code and the naming convention is used to distinguish them from existing naming conventions. The '$' sign is not allowed in names in Ruby or in C#, but it is allowed in CIL.

The *nil* attribute in *Object* is a static singleton object of type *NilClass*. In Ruby, only one object of type *NilClass* can exist, so when such an object is required it is always the singleton object that is used. By placing *nil* in *Object*, it is always available when needed and in this implementation it is used by *puts* which always returns *nil* after execution.

The *IObject* in the RubySharp class hierarchy is an interface, defining methods that are needed to be implemented by *Object*. All objects that are instantiated, sent as arguments or returned from a method, must be of type *IObject* (compare #Smalltalk's *Root* class) to mimic Ruby behaviour. By declaring all instance methods as *virtual*, the method on the correct class is always executed even though all objects are of type *IObject* (Listing 16).

```
IObject s = String._new("Hello World!");
s.to_s();
```

Listing 16. IObject usage in C#.

When *to_s()* is called on *s*, the virtual method *to_s()* in *String* is executed.

Finally, the method *method_missing* in Ruby's *Object* has also been implemented in RubySharp's *Object* class. *Method_missing* defines how calls to instance methods that do not exist are handled. When *method_missing* is used in a program, RubySharp explicitly creates the call to *method_missing*, sending in the correct arguments. In the case where it is overridden, RubySharp handles the compilation similar to the compilation of other methods.

4.2 Goal Programs

The goal programs (Appendix B) which are defined for RubySharp introduce basic Ruby concepts to be compiled into CIL. To be able to compile the goal programs, the RubySharp class hierarchy (Figure 4) was developed. It provides sufficient support for correct execution of the goal programs on the CLR. The goal programs were then translated to CIL, forming a specification for RubySharp on what CIL source code to generate from Ruby source code. In Appendix B, all four goal programs together with their CIL implementation is shown. The first goal program is shown in Listing 17 and will also be used to exemplify the compilation process described in Section 4.3.

```
puts "Hello World!"
```

Listing 17. Goal program 1.

The four goal programs introduce important Ruby concepts and rely on the existence of the RubySharp class hierarchy. In the first goal program the introduced concepts are: the creation of a *String* object, the initialization of an *IObject* array (puts takes an array of *IObjects* as argument) and sending the array as an argument to *puts*. Printing a string to the console and handling a return value, in this case the singleton object *nil*, is also introduced.

The second goal program introduces the definition of classes and methods, instantiation and the invocation of a method on that instance.

By implementing the third goal program, the concept of opening up a class and redefining an existing method is introduced.

Finally, in the fourth goal program inheritance is introduced. An instance is created of the derived class to invoke methods on both the derived class and the super class respectively.

4.3 The Compiler

The developed RubySharp compiler consists of three parts, a parser, a code generator and a CIL backend. They are described in the forthcoming paragraphs.

Parsing [16] is the first step in the compilation process. As input, the parser takes a program's source code, examining it line by line, analyzing them syntactically and semantically according to grammatical rules which are specific for every programming language. The parser builds up a representation of the program in memory as an AST (Abstract Syntax Tree), which is the parser's output. The AST is used in the remaining compilation and code generation process.

For the RubySharp compiler the Rockit [17] parser is used which is written in Ruby. The parser is currently under development and the version used in RubySharp is Rockit version 0.5.0. The AST representation of goal program 1 (Listing 17) that Rockit produces is shown in Listing 18 and in diagram form in Figure 5.

```
Program
  [
    [Call
      [nil,"puts",[StringLiteral
        ["Hello World!"]]]]]
```

Listing 18. AST of goal program 1.

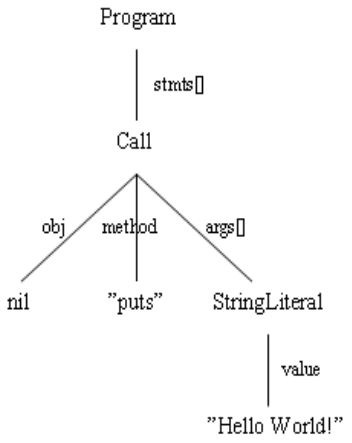

Figure 5. AST diagram of goal program 1.

The AST is built up of nodes, each containing instances of classes defined by the parser. Each class also contains a definition for its children. The top node, the class *Program* contains the complete parsed source code and also defines its child, the *stmts[]* array which contains the body of the program in subtrees. The other classes defined in this AST are *Call* and *StringLiteral*. Each of these classes are not only the top node of their own subtree, but also defines the children within it, just as *Program* defines *stmts[]*.

In goal program 1, only one statement exists, and that is *Call*. The *Call* class defines three children: *obj*, *method* and *args[]*. *Obj* is the receiver of the call which in this case is *nil*. In the current version of Rockit *nil* denotes that the method will be invoked on *self*. As an example, the receiver for other programs might be an instance of a class. The sub node *method*, stores the name of the method to be called as a Ruby *String*, in this case *puts*. The last sub node of *Call* is *args[]*, an array storing all arguments sent to *puts*. In this program only one argument exists, an instance of the class *StringLiteral* which only defines one child, *value* containing the actual string.

RubySharp traverses through the subtrees evaluating them from left to right. However, within each subtree the evaluation order might be different depending on the CIL source code to be generated from each child. To generate CIL source code from an AST, all present classes in the tree must define a method generating specific CIL code. The CIL code depends on the contents of the children and RubySharp defines these methods by adding the method *generate_to_il* to each class.

Goal program 1 is used to exemplify the compilation process and in Listing 19, the CIL code representing goal program 1 (Listing 17) is shown.

```
  1.  .locals init (class Ruby.IObject[] $0)
  2.  ldc.i4 1
  3.  newarr Ruby.IObject
  4.  stloc $0
  5.  ldloc $0
  6.  ldc.i4 0
  7.  ldstr "Hello World!"
  8.  call class Ruby.IObject
Ruby.String::_new(string)
  9.  stelem.ref
  10. ldloc $0
  11. call class Ruby.IObject
Ruby.Object::puts (class Ruby.IObject[])
  12. pop
```

Listing 19. CIL code for goal program 1.

The CIL source code in Listing 19 is based on the existence of the RubySharp class hierarchy (Appendix A) for correct execution. When compiled, this code is placed in the *Internal$Entrypoint* method. The following paragraphs give a simple explanation of the compilation process going from the AST (Listing 18, Figure 5) to the CIL source code (Listing 19).

When evaluating the AST, RubySharp starts with the *Program* node. The program node generates CIL code such as assembly header information, and for the current version of RubySharp the complete RubySharp class hierarchy is generated. The next step is to iterate through *stmts[]* generating code subtree by subtree. Since only one subtree exists in goal program 1 which top node is *Call*, RubySharp concludes that the generated CIL code should be added to *Internal$Entrypoint* method since it is outside the scope of a user defined class.

The first step when evaluating the *Call* subtree is to check which method to call using its child *method*. When encountering *puts* RubySharp knows that an *IObject[]* array must be created and initialized because *puts* takes an *IObject[]* array as argument. Even though no local variables are defined in Ruby, the generated CIL code must contain *.locals* variables for arguments because it must be possible to load arguments back onto the stack after manipulation, since objects can be removed from the stack depending on the CIL instructions performed. In this case, the instructions *stloc*, *stelem.ref* and *pop* remove the object from the stack after execution. By checking the length of *args[]* RubySharp determines how many arguments to declare. Since only one argument exists, line number 1

in Listing 19 is generated declaring only one local variable. Subsequently, line 2 adds the number one as an *int32* [1] onto the stack. On line 3, RubySharp generates code for creating the *IObject[]* array now having the size of one. On line 4, it is stored to the local variable '$0'. The naming convention for variables is to use the variable name from the Ruby source code and in this case where no variables exists, to name the needed variables starting with a '$' sign followed by a number.

When '$0' is stored it is taken of the stack, so to be able to add elements to it, code is generated to load it back onto the stack (line 5). Since the array is un-initialized, code is generated to load the first empty slot (line 6), by loading a zero onto the stack. RubySharp iterates over the arguments in the *args[]* array finding the existing argument, a *StringLiteral*. RubySharp retrieves the string through the child *value* loading it onto the stack (line 7). Since it is specified that all objects should be of the interface type *IObject*, code is generated for creating the RubySharp *String* object by calling the wrapped constructor returning an *IObject* (line 8). Finally, the *String* object is stored in the array by the generated code on line 9.

The final remaining task, before the compilation of goal program 1 is complete, is the actual call to *puts*. Hence, RubySharp generates code for loading the initialized *IObject[]* array onto the stack (line 10). RubySharp generates the CIL code for calling *puts* (line 11), sending the *IObject* array placed on the stack as argument. Since *puts* has not been overridden in the parsed Ruby program, RubySharp expects the return type to be *nil*, generating code to pop the returned object from the stack (line 12).

The compilation process for the remaining three goal programs is similar to the previously described compilation process. Each of the remaining goal programs requires the adding of the method *generate_to_il*, to classes defined in the Rockit [17] parser, e.g. *Klass* which defines how to generate CIL code for class definitions, and *LVar* defining code generation for local variables. In Appendix B, all four goal programs together with their AST representations and the CIL code they represent is shown.

The third and final part of the RubySharp compiler is the CIL backend. It contains methods for generating and adding CIL code which is stored in the backend. When the evaluation of the AST is completed and the complete CIL source code is stored in the backend, the backend saves the generated CIL code to a text file, which is compiled using the *ilasm.exe* (IL Assembler) [12] compiler.

Besides using *ilasm.exe* [12] for compilation, two additional alternatives exist for the backend. The first alternative is to use the *System.Reflection.Emit* [7] API building up a dynamic assembly in memory. When the assembly is fully created it can either be saved to disk as a static assembly (binary .exe or .dll file) or run as a dynamic assembly directly from memory. To access the *System.Reflection.Emit* [7] API from RubySharp, a Ruby/.NET Bridge [18] can be used. The bridge makes it possible to access .NET objects from Ruby and vice-versa.

Finally, the second alternative for the RubySharp backend is to create the PE file itself. When compiled, the CIL source code is stored in hexadecimal byte arrays pointed to by the metadata items defining the code. Instead of directly compiling CIL source code or generating assemblies, RubySharp can create the PE file structure by directly generating the hexadecimal byte arrays storing them statically in the PE format.

5. Discussion

When combining the concepts of Ruby and CIL, several solutions might exist for implementation. In Section 4, implementation details for RubySharp have been presented. The choices made during the development together with advantages and disadvantages are presented in this section.

In the current version of RubySharp, the complete RubySharp class hierarchy (Figure 4) is compiled into all programs. This leads to an overhead because classes and methods which are not used within the program are compiled. The overhead can be held to a minimum by only compiling the classes and methods which are used. To accomplish this, the pattern matching functionality in the Rockit [17] parser can be used. With pattern matching all method calls within an AST can be searched, adding the called methods for compilation. Pattern matching can also be used for adding mixed-in methods from modules.

Classes can be compiled in two ways by RubySharp. First, all classes with multiple declarations can be combined before compilation by using pattern matching. Second, since CIL supports *class amendment* classes can be added as they are evaluated in the AST. However, RubySharp must keep track of overridden methods for correct implementation in CIL.

The conversion between *SmallIntegers* and *LargeIntegers* in #Smalltalk is provided with the class *LargeInteger* in the *LargeInteger.dll* file. By further developing the RubySharp class hierarchy, the support of conversion between Ruby's *Fixnum* and *Bignum* can be implemented within these classes themselves, making it unnecessary to use an extern *.dll* file.

The interface *IObject* used in the RubySharp class hierarchy has two advantages. First, it offers a solution to Ruby's dynamic types. By having all instances,

arguments and return types as instances of *IObject*, Ruby behaviour can be implemented in .NET. Second, in future versions of RubySharp the possibility to inherit from .NET classes will be added, leading to a program having objects of two types, one type inheriting .NET and the other being a part of the RubySharp class hierarchy. To solve this problem, the *IObject* interface can be used to specify methods that need to be implemented on classes derived from .NET still keeping the advantages of the RubySharp class hierarchy.

The alternatives existing for the CIL backend offers both advantages and disadvantages. By generating CIL source code, programs are generated to a human readable form but provide low flexibility on dynamic features such as redefining methods and adding code during runtime. By using the *System.Reflection.Emit* [7] API, *MethodRental.SwapMethodBody* [15] and the Ruby/.NET bridge [18], RubySharp provides a solution to the dynamic features when run as a dynamic assembly, but these features cannot be saved statically. The third alternative increases complexity but offers more control on how the assemblies are stored. The CIL backend created for the current version of RubySharp is temporary just to get the proof-of-concept compiler working. For future versions, one of the two later alternatives will be used.

6. Future Work

Since RubySharp is a proof-of-concept compiler it does not offer a full implementation of the Ruby programming language. RubySharp's class hierarchy does not contain a full implementation of the classes *Object*, *String*, *NilClass* and the module *Kernel*, as well as all other Ruby classes [19].

RubySharp's code generator will be updated to use the pattern matching features of the Rockit [17] parser. It will ease the compilation of e.g. multiple class declarations where classes are reopened to add contents or when methods are overridden. It will also support compilation of only the parts in the RubySharp class hierarchy which are needed to execute the program. Furthermore, it is necessary to implement the calculation of the maxstack value for CIL methods to avoid runtime errors during execution.

The backend used by RubySharp is not optimal and implemented temporarily for the current version. It will be replaced, most likely to a backend creating the PE file directly using hexadecimal byte arrays.

Furthermore, the *System.Reflection* [7] and *System.Reflection.Emit* [7] API needs to be further examined to see what kind of support exist for e.g. Ruby's *introspection* and *eval* features.

Finally, investigating the new features of the C# 2.0 (Whidbey) release [20, 21], might provide ideas on possible solutions on how to implement Ruby's dynamic features. The new features are e.g. generics, anonymous methods, iterators and partial classes [20, 21]. Especially generic types and anonymous methods might benefit RubySharp. Generic types might be useful in the same way that the *IObject* interface is used for all arguments and return types, and anonymous methods for implementing Ruby blocks in .NET. #Smalltalk implements blocks as classes using exceptions for block returns (Section 3.2) and anonymous methods might provide a more efficient solution.

7. Conclusions

The primary goal of RubySharp was to develop a proof-of-concept compiler, able to compile a subset of the Ruby programming language into CIL, and to understand the runtime support needed for executing Ruby programs on the CLR.

A set of four goal programs were defined in Ruby to introduce basic Ruby concepts such as: printing strings, defining classes and methods, overriding methods, instantiation of classes and inheritance. Currently, the RubySharp compiler is able to compile three out of these four goal programs, goal program 1, 2 and 4 (Appendix B). Due to a design choice, RubySharp will implement method overriding (defined in goal program 3) using *MethodRental.SwapMethodBody* [15] postponing the implementation of delegates until required by a Ruby feature.

To be able to fully implement the goal programs mimicking the correct Ruby behaviour, a RubySharp class hierarchy has been developed in C#. The RubySharp class hierarchy was developed to support the features of the goal programs. Therefore, a subset of Ruby's basic classes and their methods has been implemented such as *String*, *NilClass* and *Object*. Other implementations outside the scope of the goal programs are the *method_missing* in *Object*.

An investigation of other implementations of dynamic programming language has led to the understanding of important concepts. Among them the usage of the interface *IObject* which type is used for all arguments and return values. For RubySharp it will ease the implementation of redefined methods where the return type is changed and also for the future implementation of cross-language inheritance.

Finally, an investigation of the dynamic features in .NET has taken place and unfortunately, the support for easy and efficient implementation of Ruby's

dynamic features to .NET is insufficient. However, .NET's *System.Reflection.Emit* [7] API and *MethodRental.SwapMethodBody* [15] has given insights on redefining methods and where possible solutions for Ruby's more dynamic features such as *introspection* and *eval* can be found.

8. Acknowledgement

I would like to give special thanks to my examiner Robert Feldt and my supervisor Richard Torkar for the opportunity to be involved in the development of RubySharp. Without your knowledge and support this thesis would not have become what it is; your help has been highly valuable and appreciated. I also wish to thank Robert Feldt for the possibility of using the Rockit parser in the development. Furthermore, I would like to thank John Brant at Refactory for answering questions about the design and implementation of the #Smalltalk compiler and Micael Ebbmar for help setting up the Cygwin environment.

9. References

[1] T. Thai & H.Q. Lam, *.NET Framework Essentials*, O'Reilly & Associates Inc., California, USA, 2002.

[2] The Refactory, Inc., #Smalltalk (Sharp Smalltalk), http://www.refactory.com/Software/SharpSmalltalk/, 2004-04-21.

[3] PyCon DC 2004 Proceedings, IronPython: A fast Python implementation for .NET and Mono, http://www.python.org/pycon/dc2004/papers/9/, 2004-04-22.

[4] IronPython, http://ironpython.com, 2004-04-22.

[5] ActiveState, Technologies, http://www.activestate.com/Corporate/Initiatives/NET/Research.html, 2004-05-09.

[6] D. Watkins, M. Hammond and B. Adams, *Programming in the .NET Environment*, Addison-Wesley, USA, 2002.

[7] J. liberty, *Programming C# - 3rd Edition*, O'Reilly & Associates Inc., California, USA, 2003.

[8] The Open Software License v 1.1, http://www.refactory.com/Software/SharpSmalltalk/License.txt, 2004-04-22.

[9] Thomas, David and Hunt, *Programming Ruby – The Pragmatic Programmer's Guide*, Addison-Wesley, UK, 2000.

[10] J. Weirich, An Invitation to Ruby, http://onestepback.org/articles/invitationtoruby/cover.html, 2004-04-23.

[11] mono::, The Mono open source project, http://www.go-mono.com/, 2004-05-14.

[12] S. Lidin, *Inside Microsoft .NET IL Assemble*, Microsoft Press, Washington, USA, 2002.

[13] J. Bock, *CIL Programming: Under the Hood of .NET*, APress, California, USA, 2002.

[14] J. Brant, Smalltalk in a .NET World (Power-point presentation), http://wiki.eranova.si/esug/DOWNLOAD/Slides/JohnBrant.ppt, 2004-04-21.

[15] MSDN, MethodRental.SwapMethodBody Method, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemReflectionEmitMethodRentalClassSwapMethodBodyTopic.asp, 2004-05-14.

[16] F.J.F. Benders, J.W. Haaring, T.H. Janssen, et al, *Compiler Construction – A Practical Approach*, Project Inger: http://sourceforge.net/projects/inger, USA, 2003.

[17] R. Feldt, rockit– Ruby O-o Compiler construction toolKIT, http://rockit.sf.net, 2004-05-15.

[18] B. Schroeder and J Pierce, Ruby/.NET Bridge, http://www.saltypickle.com/RubyDotNet, 2004-05-15.

[19] Ruby Central, Ruby Class and Library Reference, http://www.rubycentral.com/ref/, 2004-05-17.

[20] Microsoft Research, Generics for C# and .NET CLR, http://research.microsoft.com/projects/clrgen/, 2004-05-17.

[21] MSDN, Create Elegant Code with Anonymous Methods, Iterators and Partial Classes, http://msdn.microsoft.com/msdnmag/issues/04/05/C20/default.aspx, 2004-05-17.

RubySharp – A Ruby to CIL Compiler

Jan-Åke Hedström
Department of Computer Science
University of Trollhättan/Uddevalla
tds00jahe@thn.htu.se

Appendices

Appendix A – RubySharp class hierarchy in C# and CIL

The RubySharp class hierarchy has been developed in C# and extracted to CIL code by *ildasm.exe*. It is compiled into all programs compiled by RubySharp. The code presented in Appendix A is the only the code supporting the Ruby behaviour. The comment "// Program code" present in the method *Internal$Entrypoint* just shows where the executable code should be placed, e.g. the goal programs in Appendix B. Note also the difference in naming conventions in C# and CIL. In all methods Internal methods in C# the '$' character is not used since it is not allowed in C#.

C#

```
using System;

namespace Ruby
{
	public interface IObject
	{
		IObject to_s();
		string Internal_as_il_string();
	}

	public class Object : IObject
	{
		public static IObject nil = NilClass._new();

		public Object()
		{
		}

		public static IObject _new()
		{
			IObject o = new Object();
			return o;
		}

		public static IObject puts(params IObject[] obj)
		{
			foreach(IObject o in obj)
			{
				IObject str = o.to_s();
				if (str.GetType() == typeof(Ruby.String))
				{
					Console.WriteLine(str.Internal_as_il_string());
				}
				else
				{
					Console.WriteLine(o.Internal_as_il_string());
				}
			}
			return nil;
		}

		public virtual void method_missing(params IObject[] obj)
		{
			Console.WriteLine(AppDomain.CurrentDomain.FriendlyName
				+ ": undefined method '"
				+ obj[0] + "' for #<"
				+ this.Internal_as_il_string()
				+ ":" + Internal_get_address(this)
```

```
                        + "> (NoMethodError)");
                Environment.Exit(0);
        }

        public virtual IObject to_s()
        {
                IObject str = String._new(this.Internal_as_il_string());
                return str;
        }

        public virtual string Internal_as_il_string()
        {
                string s = "#<" + this.ToString() + ":" + Internal_get_address(this) + ">";
                return s;
        }

        private string Internal_get_address(IObject o)
        {
                // Exchange GetHashCode() for address of o, and
                // change method definition to private unsafe string ...
                string s = o.GetHashCode().ToString();
                return s;
        }
}

public class NilClass : Object
{
        private static IObject singleton;

        public NilClass()
        {
        }

        public static new IObject _new()
        {
                if (NilClass.singleton == null)
                {
                        NilClass.singleton = new NilClass();
                }
                return singleton;
        }
}

public class String : Object
{
        private string str;

        public static IObject _new(string s)
        {
                IObject str = new String(s);
                return str;
        }

        public String(string s)
        {
                str = s;
        }
```

```
		public override string Internal_as_il_string()
		{
			return this.str;
		}

		public override IObject to_s()
		{
			return this;
		}
	}

	public class Internal_Exec : Object
	{
		public static void Main()
		{
			// Program code
		}
	}
}
```

CIL

```
.assembly extern mscorlib{}
.assembly Test{}
.module Test.exe

.namespace Ruby
{
  .class interface public abstract auto ansi IObject
  {
    .method public hidebysig newslot abstract virtual instance class Ruby.IObject to_s() cil managed
    {
    }
    .method public hidebysig newslot abstract virtual instance string Internal$as_il_string() cil managed
    {
    }
  }
  .class public auto ansi beforefieldinit Object extends [mscorlib]System.Object implements Ruby.IObject
  {
    .field public static class Ruby.IObject nil
    .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
    {
      .maxstack 1
      ldarg.0
     call instance void [mscorlib]System.Object::.ctor()
      ret
    }
    .method public hidebysig static class Ruby.IObject _new() cil managed
    {
      .maxstack 1
      .locals init (class Ruby.IObject V_0, class Ruby.IObject V_1)
     newobj instance void Ruby.Object::.ctor()
     stloc.0
     ldloc.0
     stloc.1
     br.s IL_0001
     IL_0001: ldloc.1
     ret
    }
```

```
  .method public hidebysig static class Ruby.IObject puts(class Ruby.IObject[] obj) cil managed
   {
    .param [1]
    .custom instance void [mscorlib]System.ParamArrayAttribute::.ctor() = ( 01 00 00 00 )
    .maxstack 2
    .locals init (class Ruby.IObject V_0, class Ruby.IObject V_1, class Ruby.IObject V_2, class Ruby.IObject[] V_3, int32 V_4)
    ldarg.0
    stloc.3
    ldc.i4.0
    stloc.s V_4
    br.s IL_0001
    IL_0002: ldloc.3
    ldloc.s V_4
    ldelem.ref
    stloc.0
    ldloc.0
    callvirt instance class Ruby.IObject Ruby.IObject::to_s()
    stloc.1
    ldloc.1
    callvirt instance class [mscorlib]System.Type [mscorlib]System.Object::GetType()
    ldtoken Ruby.String
    call class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle(valuetype [mscorlib]System.RuntimeTypeHandle)
    bne.un.s IL_0003
    ldloc.1
    callvirt instance string Ruby.IObject::Internal$as_il_string()
    call void [mscorlib]System.Console::WriteLine(string)
    br.s IL_0005
    IL_0003: ldloc.0
    callvirt instance string Ruby.IObject::Internal$as_il_string()
    call void [mscorlib]System.Console::WriteLine(string)
    IL_0005:  ldloc.s   V_4
    ldc.i4.1
    add
    stloc.s V_4
    IL_0001: ldloc.s V_4
    ldloc.3
    ldlen
    conv.i4
    blt.s IL_0002
    ldsfld class Ruby.IObject Ruby.Object::nil
    stloc.2
    br.s IL_0004
    IL_0004: ldloc.2
    ret
   }
   .method public hidebysig newslot virtual instance void method_missing(class Ruby.IObject[] obj) cil managed
   {
    .param [1]
    .custom instance void [mscorlib]System.ParamArrayAttribute::.ctor() = ( 01 00 00 00 )
    .maxstack  4
    .locals init (object[] V_0)
    ldc.i4.8
    newarr [mscorlib]System.Object
    stloc.0
    ldloc.0
```

```
  ldc.i4.0
  call class [mscorlib]System.AppDomain [mscorlib]System.AppDomain::get_CurrentDomain()
  callvirt   instance string [mscorlib]System.AppDomain::get_FriendlyName()
  stelem.ref
  ldloc.0
  ldc.i4.1
  ldstr ": undefined method '"
  stelem.ref
  ldloc.0
  ldc.i4.2
  ldarg.1
  ldc.i4.0
  ldelem.ref
  stelem.ref
  ldloc.0
  ldc.i4.3
  ldstr "' for #<"
  stelem.ref
  ldloc.0
  ldc.i4.4
  ldarg.0
  callvirt instance string Ruby.Object::Internal$as_il_string()
  stelem.ref
  ldloc.0
  ldc.i4.5
  ldstr ":"
  stelem.ref
  ldloc.0
  ldc.i4.6
  ldarg.0
  ldarg.0
  call instance string Ruby.Object::Internal$get_address(class Ruby.IObject)
  stelem.ref
  ldloc.0
  ldc.i4.7
  ldstr "> (NoMethodError)"
  stelem.ref
  ldloc.0
  call string [mscorlib]System.String::Concat(object[])
  call void [mscorlib]System.Console::WriteLine(string)
  ldc.i4.0
  call void [mscorlib]System.Environment::Exit(int32)
  ret
 }
 .method public hidebysig newslot virtual instance class Ruby.IObject to_s() cil managed
 {
  .maxstack  1
  .locals init (class Ruby.IObject V_0, class Ruby.IObject V_1)
  ldarg.0
  callvirt instance string Ruby.Object::Internal$as_il_string()
  call class Ruby.IObject Ruby.String::_new(string)
  stloc.0
  ldloc.0
  stloc.1
  br.s IL_0001
  IL_0001: ldloc.1
  ret
 }
```

```
.method public hidebysig newslot virtual instance string Internal$as_il_string() cil managed
{
  .maxstack 4
  .locals init (string V_0, string V_1, string[] V_2)
  ldc.i4.5
  newarr [mscorlib]System.String
  stloc.2
  ldloc.2
  ldc.i4.0
  ldstr "#<"
  stelem.ref
  ldloc.2
  ldc.i4.1
  ldarg.0
  callvirt instance string [mscorlib]System.Object::ToString()
  stelem.ref
  ldloc.2
  ldc.i4.2
  ldstr ":"
  stelem.ref
  ldloc.2
  ldc.i4.3
  ldarg.0
  ldarg.0
  call instance string Ruby.Object::Internal$get_address(class Ruby.IObject)
  stelem.ref
  ldloc.2
  ldc.i4.4
  ldstr ">"
  stelem.ref
  ldloc.2
  call string [mscorlib]System.String::Concat(string[])
  stloc.0
  ldloc.0
  stloc.1
  br.s IL_0001
  IL_0001: ldloc.1
  ret
}
.method private hidebysig instance string Internal$get_address(class Ruby.IObject o) cil managed
{
  .maxstack 1
  .locals init (string V_0, string V_1, int32 V_2)
  ldarg.1
  callvirt instance int32 [mscorlib]System.Object::GetHashCode()
  stloc.2
  ldloca.s V_2
  call instance string [mscorlib]System.Int32::ToString()
  stloc.0
  ldloc.0
  stloc.1
  br.s IL_0001
  IL_0001: ldloc.1
  ret
}
.method private hidebysig specialname rtspecialname static void .cctor() cil managed
{
  .maxstack 1
```

```
    call class Ruby.IObject Ruby.NilClass::_new()
    stsfld class Ruby.IObject Ruby.Object::nil
    ret
  }
}
.class public auto ansi beforefieldinit String extends Ruby.Object
{
  .field private string str
  .method public hidebysig static class Ruby.IObject _new(string s) cil managed
  {
    .maxstack 2
    .locals init (class Ruby.IObject V_0, class Ruby.IObject V_1)
    ldarg.0
    newobj instance void Ruby.String::.ctor(string)
    stloc.0
    ldloc.0
    stloc.1
    br.s IL_0001
    IL_0001: ldloc.1
    ret
  }
  .method public hidebysig specialname rtspecialname instance void .ctor(string s) cil managed
  {
    .maxstack 2
    ldarg.0
    call instance void Ruby.Object::.ctor()
    ldarg.0
    ldarg.1
    stfld string Ruby.String::str
    ret
  }
  .method public hidebysig virtual instance string Internal$as_il_string() cil managed
  {
    .maxstack 1
    .locals init (string V_0)
    ldarg.0
    ldfld string Ruby.String::str
    stloc.0
    br.s IL_0001
    IL_0001: ldloc.0
    ret
  }
  .method public hidebysig virtual instance class Ruby.IObject to_s() cil managed
  {
    .maxstack 1
    .locals init (class Ruby.IObject V_0)
    ldarg.0
    stloc.0
    br.s IL_0001
    IL_0001: ldloc.0
    ret
  }
}
.class public auto ansi beforefieldinit NilClass extends Ruby.Object
{
  .field private static class Ruby.IObject singleton
  .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
  {
```

```
    .maxstack 1
    ldarg.0
    call instance void Ruby.Object::.ctor()
    ret
   }
   .method public hidebysig static class Ruby.IObject _new() cil managed
   {
    .maxstack 1
    .locals init (class Ruby.IObject V_0)
    ldsfld class Ruby.IObject Ruby.NilClass::singleton
    brtrue.s IL_0001
    newobj instance void Ruby.NilClass::.ctor()
    stsfld class Ruby.IObject Ruby.NilClass::singleton
    IL_0001: ldsfld class Ruby.IObject Ruby.NilClass::singleton
    stloc.0
    br.s IL_0002
    IL_0002: ldloc.0
    ret
   }
  }
  .class public auto ansi beforefieldinit Internal$Exec extends Ruby.Object
  {
   .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
   {
    .maxstack 1
    ldarg.0
    call instance void Ruby.Object::.ctor()
    ret
   }
   .method public hidebysig static void Internal$Entrypoint() cil managed
   {
    .entrypoint
    // Program code
    ret
   }
  }
}
```

Appendix B – Goal Programs, their AST Representation and CIL Source Code

In Appendix B all four goal programs are presented. First, the Ruby source code is shown, followed by the AST (Abstract Syntax Tree) representation produced by the Rockit parser. Finally, the CIL (Common Intermediate Language) is shown. The CIL code is only the code that represents the Ruby source code. For a fully executable program, the RubySharp class hierarchy in Appendix A must be added.

Goal Program 1: Printing a string to the console

Ruby:

```
puts "Hello World!"
```

AST:

```
Program[[Call[nil, "puts", [StringLiteral["Hello World!"]]]]]
```

CIL:

```
.maxstack  3
.locals init (class Ruby.IObject[] V_0)
ldc.i4.1
newarr Ruby.IObject
stloc.0
ldloc.0
ldc.i4.0
ldstr "Hello RubySharp!"
call class Ruby.IObject Ruby.String::_new(string)
stelem.ref
ldloc.0
call class Ruby.IObject Ruby.Object::puts(class Ruby.IObject[])
pop
ret
```

Goal Program 2: Class and method definition

Ruby:

```
class C
  def m1
    puts "Hello RubySharp!"
  end
end

C.new.m1                    // Output: Hello RubySharp!
```

AST:

```
Program[
 [Klass[
   Const["C"],
   nil,
   [Def["m1", [], [Call[nil, "puts", [StringLiteral["Hello RubySharp!"]]]]]]],
  Call[Call[Const["C"], "new", []], "m1", []]]]
```

CIL:

```
.class public auto ansi beforefieldinit C extends Ruby.Object
{
  .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
  {
   .maxstack  1
   ldarg.0
   call instance void Ruby.Object::.ctor()
   ret
  }
  .method public hidebysig static class Ruby.C _new() cil managed
  {
   .maxstack 1
   .locals init (class Ruby.C V_0)
   newobj instance void Ruby.C::.ctor()
   stloc.0
   ldloc.0
   ret
  }
  .method public hidebysig instance void m1() cil managed
  {
   .maxstack 3
   .locals init (class Ruby.IObject[] V_0)
   ldc.i4.1
   newarr Ruby.IObject
   stloc.0
   ldloc.0
   ldc.i4.0
   ldstr "Hello RubySharp!"
   call class Ruby.IObject Ruby.String::_new(string)
   stelem.ref
   ldloc.0
   call class Ruby.IObject Ruby.Object::puts(class Ruby.IObject[])
   pop
   ret
  }
}
.class public auto ansi beforefieldinit Internal$Exec extends Ruby.Object
{
  .method public hidebysig static void Internal$Entrypoint() cil managed
  {
   .entrypoint
   .maxstack 1
   call class Ruby.C Ruby.C::_new()
   callvirt instance void Ruby.C::m1()
   ret
  }
  .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
  {
   .maxstack  1
   ldarg.0
   call  instance void Ruby.Object::.ctor()
   ret
  }
}
```

Goal Program 3: Overriding methods

Ruby:

```
class C
  def m1
    puts "Original m1!"
  end
end

C.new.m1                    // Output: Original m1!

class C
  def m1
    puts "Replaced m1!"
  end
end

C.new.m1                    // Output: Replaced m1!
```

AST:

```
Program[
 [Klass[
   Const["C"],
   nil,
   [Def["m1", [], [Call[nil, "puts", [StringLiteral["Original m1!"]]]]]]],
  Call[Call[Const["C"], "new", []], "m1", []],
  Klass[
   Const["C"],
   nil,
   [Def["m1", [], [Call[nil, "puts", [StringLiteral["Replaced m1!"]]]]]]],
  Call[Call[Const["C"], "new", []], "m1", []]]]
```

CIL:

```
.class public auto ansi beforefieldinit C extends Ruby.Object
{
   .class auto ansi sealed nested public Del extends [mscorlib]System.MulticastDelegate
   {
      .method public hidebysig specialname rtspecialname instance void .ctor(object 'object', native int 'method') runtime managed
      {
      }
      .method public hidebysig virtual instance void Invoke() runtime managed
      {
      }
      .method public hidebysig newslot virtual
            instance class [mscorlib]System.IAsyncResult
            BeginInvoke(class [mscorlib]System.AsyncCallback callback,
                  object 'object') runtime managed
      {
      }
      .method public hidebysig newslot virtual instance void EndInvoke(class [mscorlib]System.IAsyncResult result) runtime managed
      {
      }
   }
   .field private static bool _delegate_overridden
```

```
.field public class Ruby.C/Del m1
.method public hidebysig specialname rtspecialname instance void  .ctor() cil managed
{
  .maxstack 4
  ldarg.0
  call instance void Ruby.Object::.ctor()
  ldsfld bool Ruby.C::_delegate_overridden
  brtrue.s IL_0001
  ldarg.0
  ldarg.0
  ldftn      instance void Ruby.C::_old_m1()
  newobj     instance void Ruby.C/Del::.ctor(object, native int)
  stfld class Ruby.C/Del Ruby.C::m1
  br.s  IL_0002
  ldarg.0
  ldarg.0
  ldftn instance void Ruby.C::_new_m1()
  newobj  instance void Ruby.C/Del::.ctor(object, native int)
  stfld class Ruby.C/Del Ruby.C::m1
  IL_0002:  ret
}
.method public hidebysig static class Ruby.C _new() cil managed
{
  .maxstack 1
  .locals init (class Ruby.C V_0)
  newobj     instance void Ruby.C::.ctor()
  stloc.0
  ldloc.0
  ret
}
.method public hidebysig instance void _override_delegate() cil managed
{
  .maxstack 4
  ldc.i4.1
  stsfld bool Ruby.C::_delegate_overridden
  ldarg.0
  ldarg.0
  ldftn instance void Ruby.C::_new_m1()
  newobj instance void Ruby.C/Del::.ctor(object, native int)
  stfld class Ruby.C/Del Ruby.C::m1
  ret
}
.method public hidebysig instance void _old_m1() cil managed
{
  .maxstack  3
  .locals init (class Ruby.IObject[] V_0)
  ldc.i4.1
  newarr Ruby.IObject
  stloc.0
  ldloc.0
  ldc.i4.0
  ldstr "Original m1"
  call class Ruby.IObject Ruby.String::_new(string)
  stelem.ref
  ldloc.0
  call class Ruby.IObject Ruby.Object::puts(class Ruby.IObject[])
  pop
  ret
```

```
  }
  .method public hidebysig instance void _new_m1() cil managed
  {
    .maxstack  3
    .locals init (class Ruby.IObject[] V_0)
    ldc.i4.1
    newarr     Ruby.IObject
    stloc.0
    ldloc.0
    ldc.i4.0
    ldstr "Overloaded m1"
    call class Ruby.IObject Ruby.String::_new(string)
    stelem.ref
    ldloc.0
    call class Ruby.IObject Ruby.Object::puts(class Ruby.IObject[])
    pop
    ret
  }
  .method private hidebysig specialname rtspecialname static void .cctor() cil managed
  {
    .maxstack 1
    ldc.i4.0
    stsfld bool Ruby.C::_delegate_overridden
    ret
  }
}
.class public auto ansi beforefieldinit Internal$Exec extends Ruby.Object
{
  .method public hidebysig static void Internal$Entrypoint() cil managed
  {
    .entrypoint
    .maxstack  1
    call class Ruby.C Ruby.C::_new()
    ldfld class Ruby.C/Del Ruby.C::m1
    callvirt instance void Ruby.C/Del::Invoke()
    call class Ruby.C Ruby.C::_new()
    callvirt instance void Ruby.C::_override_delegate()
    call class Ruby.C Ruby.C::_new()
    ldfld class Ruby.C/Del Ruby.C::m1
    callvirt instance void Ruby.C/Del::Invoke()
    ret
  }
  .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
  {
    .maxstack 1
    ldarg.0
    call instance void Ruby.Object::.ctor()
    ret
  }
}
```

Goal Program 4: Inheritance

Ruby:

```
class Super
  def s1
    puts "s1!"
  end
end

class C < Super
  def m1
    puts "m1!"
  end
end

c = C.new
c.s1              // Output: s1!
c.m1              // Output: m1!
```

AST:

```
Program[
 [Klass[
   Const["Super"],
   nil,
   [Def["s1", [], [Call[nil, "puts", [StringLiteral["s1!"]]]]]]],
  Klass[
   Const["C"],
   Const["Super"],
   [Def["m1", [], [Call[nil, "puts", [StringLiteral["m1!"]]]]]],
  Assign[LVar["c"], Call[Const["C"], "new", []]],
  Call[LVar["c"], "s1", []],
  Call[LVar["c"], "m1", []]]]
```

CIL:

```
.class public auto ansi beforefieldinit Super extends Ruby.Object
{
  .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
  {
    .maxstack  1
    ldarg.0
    call instance void Ruby.Object::.ctor()
    ret
  }
  .method public hidebysig static class Ruby.Super _new() cil managed
  {
    .maxstack  1
    .locals init (class Ruby.Super V_0)
    newobj     instance void Ruby.Super::.ctor()
    stloc.0
    ldloc.0
    ret
  }
  .method public hidebysig instance void s1() cil managed
  {
    .maxstack 3
```

```
      .locals init (class Ruby.IObject[] V_0)
      ILldc.i4.1
      newarr Ruby.IObject
      stloc.0
      ldloc.0
      ldc.i4.0
      ldstr "s1"
      call class Ruby.IObject Ruby.String::_new(string)
      stelem.ref
      ldloc.0
      call class Ruby.IObject Ruby.Object::puts(class Ruby.IObject[])
      pop
      ret
    }
}
.class public auto ansi beforefieldinit C extends Ruby.Super
{
    .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
    {
      .maxstack 1
      ldarg.0
      call instance void Ruby.Super::.ctor()
      ret
    }
    .method public hidebysig static class Ruby.C _new() cil managed
    {
      .maxstack 1
      .locals init (class Ruby.C V_0)
      newobj instance void Ruby.C::.ctor()
      stloc.0
      ldloc.0
      ret
    }
    .method public hidebysig instance void m1() cil managed
    {
      .maxstack 3
      .locals init (class Ruby.IObject[] V_0)
      ldc.i4.1
      newarr Ruby.IObject
      stloc.0
      ldloc.0
      ldc.i4.0
      ldstr "m1"
      call class Ruby.IObject Ruby.String::_new(string)
      stelem.ref
      ldloc.0
      call class Ruby.IObject Ruby.Object::puts(class Ruby.IObject[])
      pop
      ret
    }
}
.class public auto ansi beforefieldinit Internal$Exec extends Ruby.Object
{
    .method public hidebysig static void Internal$Entrypoint() cil managed
    {
      .entrypoint
      .maxstack 1
      .locals init (class Ruby.C V_0)
```

```
    call class Ruby.C Ruby.C::_new()
    stloc.0
    ldloc.0
    callvirt instance void Ruby.Super::s1()
    ldloc.0
    callvirt instance void Ruby.C::m1()
    ret
   }
   .method public hidebysig specialname rtspecialname instance void .ctor() cil managed
   {
    .maxstack 1
    ldarg.0
    call instance void Ruby.Object::.ctor()
    ret
   }
}
```