Efficient Virtual Machine Support of Runtime Structural Reflection

Francisco Ortin [a,*], Jose Manuel Redondo [a],
J. Baltasar García Perez-Schofield [b]

[a] *University of Oviedo, Computer Science Department,Calvo Sotelo s/n, 33007, Oviedo, Spain*

[b] *University of Vigo, Computer Science Department, As Lagoas s/n, 32004, Ourense, Spain*

Abstract

Increasing trends towards adaptive, distributed, generative and pervasive software have made object-oriented dynamically typed languages become increasingly popular. These languages offer dynamic software evolution by means of reflection, facilitating the development of dynamic systems. Unfortunately, this dynamism commonly imposes a runtime performance penalty. In this paper, we describe how to extend a production JIT-compiler virtual machine to support runtime object-oriented structural reflection offered by many dynamic languages. Our approach improves runtime performance of dynamic languages running on statically-typed virtual machines. At the same time, existing statically-typed languages are still supported by the virtual machine.

We have extended the .Net platform with runtime structural reflection adding prototype-based object-oriented semantics to the statically typed class-based model of .Net, supporting both kinds of programming languages. The assessment of runtime performance and memory consumption has revealed that a direct support of structural reflection in a production JIT-based virtual machine designed for statically typed languages provides a significant performance improvement for dynamically typed languages.

Key words: Structural reflection, dynamically typed languages, JIT compilation, SSCLI, virtual machine, prototype-based object-oriented model

* Corresponding author.
Email addresses: ortin@lsi.uniovi.es (Francisco Ortin), redondojose@uniovi.es (Jose Manuel Redondo), jgarcia@uvigo.es (J. Baltasar García Perez-Schofield).
URLs: http://www.di.uniovi.es/∼ortin/ (Francisco Ortin),

1 Introduction

Object-oriented dynamically typed languages like Python [1], Ruby [2], Dylan [3], Lua [4] or Groovy [5] are becoming increasingly popular for developing different kinds of applications such as Web development, game scripting, interactive programming, rapid prototyping, dynamic aspect-oriented programming, and any kind of software that requires dynamic adaptiveness. These languages build on the Smalltalk idea of supporting reasoning about (and customizing) program structure, behavior and environment at runtime. That is the reason why this trend is commonly referred to as *the revival of dynamic languages* [6].

The main objective of dynamically typed languages is to model the dynamicity that is sometimes required for building highly context-dependent software, due to the mobility of both the software itself and its users. Features such as metaprogramming, reflection, mobility, dynamic reconfiguration and distribution are the domain of these languages. These benefits of dynamism are offset by the lack of static type checking and a considerable runtime performance penalty.

Looking for code portability, dynamically typed languages commonly execute programs using a virtual machine. Implementing virtual machines as interpreters involves a significant performance penalty compared to native code execution. Consequently, there has been considerable research aimed at improving the performance of virtual machines. Techniques like virtual machine Just In Time (JIT) compilation [7] or runtime adaptive optimization [8] have reached such maturity that many vendors distribute this kind of platforms as appropriate to implement efficient applications. Nowadays, commercial languages like Java or C#, widely used in software development, are compiled down to virtual machines.

In order to improve runtime performance of existing implementations of dynamically-typed object-oriented languages, our work has been focused on applying the same techniques that have made virtual machines a valid alternative to develop commercial software. Therefore, we have extended a production virtual machine JIT-compiler to evaluate whether it is suitable for improving runtime performance of existing implementations of these languages. Moreover, extending a widely used virtual machine entails maintaining the support of existing languages compiled down to this machine.

The main contribution of this work is the design, implementation and evaluation of an extension of a commercial JIT-based virtual machine designed for

---

`http://www.di.uniovi.es/~redondojose/` (Jose Manuel Redondo),
`http://webs.uvigo.es/jbgarcia/` (J. Baltasar García Perez-Schofield).

non-reflective static languages in order to efficiently support runtime structural reflection of object-oriented dynamically typed languages. The resulting system not only obtains a significant improvement of dynamically typed object-oriented languages implementations, but also supports the execution of any existing .NET programming language.

The rest of this paper is structured as follows. In the next section, we provide motivation and relevant background. Section 3 summarizes the structure of the Microsoft Shared Source CLI implementation of the .NET platform. Section 4 discusses the design of our reflective object model, and the implementation is presented in Section 5. We assess runtime performance in Section 6, and Section 7 discusses related work. Finally, Section 8 presents the ending conclusions.

2 Motivation & Background

2.1 Object-Oriented Dynamically Typed Languages

Due to the flexibility they offer, object-oriented dynamically typed languages are becoming increasingly popular. Some examples of the success of such languages are the *Ruby on Rails* framework [9] and the *AJAX (Asynchronous JavaScript And XML)* development technique for creating Web applications [10]; the incorporation of a standard framework to allow dynamic scripting programs to be executed from, and have access to, the Java platform version 1.6 (JSR 223) [11], and the `invokedynamic` instruction to support the implementation of dynamically typed object oriented languages to be included in Java SE 1.7 [12]; the Dynamic Language Runtime (DLR), launched by Microsoft to add to the .NET platform a set of services to facilitate the implementation of dynamic languages; the wide range of dynamic aspect-oriented tools that has been built on top of dynamically typed languages [13–17]; and the Zope application server for building content management systems, intranets and custom applications [18].

2.2 Limitations of Dynamic Typing

Unlike statically typed languages (Java, C# or C++), dynamically typed ones do not perform type checking at compile time. Static typing offers the programmer the early detection of type errors, making possible to fix them immediately rather than discovering them at runtime —when the programmer's efforts might be aimed at some other task, or even after the program

has been deployed [19]. Another important limitation of existing implementations of dynamically typed languages is that their runtime performance is commonly slower than commercial static programming languages (discussed in the following section).

In order to reduce the lack of early type error detection, dynamically typed languages have been designed to be easily integrated with unit testing facilities and suites —in fact, the SUnit testing framework for Smalltalk [20] was the first implementation of the xUnit family of frameworks such as JUnit. In the case of Python, the programmer typically implements testing routines at the end of every module; it is also common to use the PyUnit test suite; finally, PyChecker is a static analysis tool that is also used for finding bugs in Python source code.

Another approach to overcome the limitations of dynamic typing is the integration of static and dynamic typing into the same language [21]. Static typing allow earlier detection of programming mistakes, better documentation, more opportunities for compiler optimizations, and increased runtime performance. Dynamic typing languages provide a solution to a kind of computational incompleteness inherent to statically typed languages, offering, for example, storage of persistent data, inter-process communication, dynamic program behavior customization, or generative programming [22]. There are situations in programming when one would like to use dynamic types even in the presence of advanced static type systems [23]. Instead of providing programmers with a black or white choice between static or dynamic typing, it could be useful to strive for softer type systems [24]. That is what Meijer and Drayton refer to as *static typing where possible, dynamic typing when needed* [21].

Using one of the approaches mentioned above, the programmer can benefit from both the robustness of statically typed languages and the flexibility dynamically type ones. That is the reason why we have focused our efforts on optimizing the implementation of dynamically typed languages.

2.3 Dynamically Typed Languages on the .NET *and Java Platforms*

Looking for code mobility, portability, and distribution facilities, dynamically typed languages are usually compiled to the intermediate language of an abstract machine. Since the computational model of these abstract machines is more complex than the one implemented by statically typed languages, it is more difficult to implement a JIT-compiler for these dynamic platforms. This inherent complexity plus the performance cost of inferring and checking types at runtime commonly involve a runtime performance penalty when compared

to statically typed languages.

Since the research done by Chambers and Ungar in *customized dynamic compilation* applied to the Self programming language [25], virtual machine implementations have become faster by optimizing the binary code generated at runtime. These optimizations were successfully applied to both Self [26] and Smalltalk [7] dynamically typed reflective programming languages. Nowadays, dynamic adaptive *HotSpot* optimizer compilers combine fast compilation and runtime optimization of those parts of the code that are most frequently executed. These techniques have made Java and .NET virtual machines a real alternative to develop many types of software products.

Taking an existing production JIT-based virtual machine (like Java or .NET) that supports statically typed languages, and enhancing it to support reflective dynamically typed languages might involve two mayor benefits: first, an important performance improvement, as demonstrated with Self [26] and Smalltalk [7]; second, the support of both dynamic and static typing in a language-neutral way.

Most works aimed at supporting reflective dynamically typed languages over the .NET and Java platforms are restricted to compilers that generate Java or .NET bytecodes. Taking Python as an example, there exist different implementations for the Microsoft .NET platform that simulate Python features (Python for .NET from the Zope Community, IronPython from Microsoft, and the Python for .NET research project from ActiveState). The implementations that use the Java Virtual Machine (Jython, formerly called JPython) follow the same approach. As we will show in Section 6, these approaches show poor runtime performance in reflective scenarios.

Although Microsoft and Sun platforms are increasingly incorporating dynamic languages features such as dynamic code generation and code instrumentation, they were created to support class-based static languages. At the moment, these platforms do not provide dynamic modification of structures of classes and objects once the application is running (dynamic structural reflection). Therefore, existing compilers of dynamically typed languages that generate code to these platforms (e.g., IronPython and Jython) must implement an additional layer to support these features, leading to a poor runtime performance [27] –we will evaluate them in Section 6.

The work presented in this paper uses a virtual machine with JIT compilation to directly support structural reflection in a language-neutral way. Unlike existing implementations, our approach is based on extending the statically typed computational model of a production virtual machine designed for statically typed languages, adding the reflective services of many dynamically typed languages. This new computational model is then translated into the

native code of a specific platform at runtime, using a JIT compiler. Instead of generating extra code to simulate the computational model of dynamically typed languages, the virtual machine will support these services directly. As a result, a significant performance improvement is achieved, and both dynamically typed and statically typed languages are supported.

2.4 Structural Reflection

Reflection is *the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions* [28]. In a reflective language, the computational domain is enhanced with its self-representation, offering at runtime its structure and semantics as computable data. Reflection has been recognized as a suitable tool to aid the dynamic evolution of running systems, being the primary technique to obtain meta-programming, adaptiveness, and dynamic reconfiguration features of dynamic languages [29]. Computational reflection is *the activity performed by a computational system when doing computation about (and by possibly affecting) its own computation* [28].

Considering that observation and modification of the system's self-representation are two aspects of reflection, we have [30]:

- **Introspection**: Self-representation of programs can be dynamically consulted but not modified. Both Java and .NET platforms offer introspection by means of the `java.lang.reflect` package (Java) and `System.Reflection` namespace (.NET). With these services, the programmer can obtain information about classes, objects, methods and fields at runtime.
- **Intercession**: The ability of a program to modify its own execution state, or alter its own interpretation or meaning. The Python capability of modifying the inheritance graph at runtime is an example of intercession.

Another criterion to categorize runtime reflective systems is taking into consideration what can be reflected. According to this condition, two levels of reflection are identified:

- **Structural Reflection**: System structure can be accessed. In case a structure is modified, changes will be reflected at runtime. An example of this kind of reflection is the Python feature of adding fields or methods to both objects and classes.
- **Behavioral Reflection**: Access to system semantics is offered. In case the semantics is modified, it will involve a customization of the runtime behavior of programs. For instance, *MetaXa* (formerly called *MetaJava* [31]) is a Java extension that offers the programmer the ability to dynamically modify the method dispatching mechanism. The most common technique to reach this level of reflection is a Meta-Object Protocol (MOP) [32].

Many dynamically typed languages provide runtime intercession at the structural level of reflection, offering a high level of adaptiveness at runtime. Much research on MOPs has revealed that behavioral reflection impose a high performance penalty in comparison with the benefits it provides, and its implementation is certainly difficult [33]. Moreover, many behavioral features could be simulated with structural reflection (e.g., adapting method invocation semantics could be simulated with a class or object method wrapping service developed with structural reflection). These are the reasons why we have implemented structural reflection services but not behavioral ones, offering a good trade-off between flexibility and efficiency.

We have focused our efforts on optimizing the structural reflective services offered by many object-oriented dynamically typed languages (such as Python or Ruby). For that purpose, we have taken a production JIT-based virtual machine that supports statically typed languages. Although its JIT compiler generates optimized native code, it difficult to support a reflective model where structures of objects and classes can be modified at runtime, because its design is highly focused on statically typed languages.

2.5 Structural Reflection and Dynamic Typing

Runtime intercession, at the structural reflection level, offers the programmer dynamic structure modification of classes and objects. Although this structure alteration is performed at runtime, it is not strictly necessary to implement it with dynamic typing. A static type system could be able to infer dynamic types at compile time. For instance, the *Fickle* language permit statically typed object re-classification (changing the type of objects at runtime) [34]. However, depending on dynamic values, an object type cannot be changed to classes whose common interface cannot be safely deduced at compile time.

Static type checking is a compile-time abstraction of the runtime behavior of programs, and hence it is necessarily only partially sound and incomplete [21]; there are programs that cannot go wrong, but they cannot be statically type-checked. For these scenarios, dynamic typing is more appropriate. As an example, the *wide classes* approach allows an object to be temporarily widened to a subclass, extending the object structure with additional members [35]. It is possible to widen an object with two disjoint sets of messages and, depending on runtime values, pass those recently added specific messages. This is possible because, unlike Fickle, they implement a dynamic type system.

Since our work is focused on building a structural reflective platform to support both dynamically and statically typed programming languages, we decided to add dynamic typing to the virtual machine. As an example, it should be

possible to pass a message to an object when that message could have been added at runtime using structural reflection, checking its suitability and the returning type at runtime. High-level programming languages may implement a reflective static type system (like the *Fickle* approach), a dynamic one (like *wide classes*), or even a hybrid approach [36].

2.6 A Motivating Example

Our objective is to extend the semantics of a virtual machine to support fully-fledged dynamic structural reflective primitives built into the platform internals, evaluating the performance benefits provided by its JIT native code generator. The statically typed object-oriented model must be maintained in order to be backward compatible.

As an example to clarify the objectives of our project, Figure 1 shows the same program written in Python and Ruby. This program modifies the structure of classes and objects at runtime, using the structural reflective primitives of both languages. Our platform should support this kind of services.

We first create a `Point` class with its constructor and the `move` and `draw` methods. An instance is then created (`point`) and a `draw` message is passed. Then, we modify the structure of a single object adding a new `z` field and its respective `draw3D` method. A new `getX` method is set to the `Point` class, making any `Point` instance capable of responding to the `getX` message. Finally, a new `isShowing` field is added to the `Point` class.

The last reflective primitive is implemented in a different way in both languages. In Python, the structure of every `Point` instance is extended with the `isShowing` field. However, Ruby interprets this operation as the addition of a new `isShowing` field to the `Point` class. In the design section, we will analyze how to support both behaviors by the same virtual machine.

3 Shared Source Common Language Infrastructure

Compiling languages to the intermediate code of a virtual machine offers many benefits such as platform neutrality, compiler simplification, application distribution, direct support of high-level paradigms and application interoperability [37]. In addition, compiling languages to a virtual machine with a lower abstraction level improves runtime performance in comparison with direct interpretation of programs.

We have used the Microsoft .NET platform as the targeted virtual machine

```python
class Point:
  "Constructor"
  def __init__(self, x, y):
    self.x=x
    self.y=y

  "Move Method"
  def move(self, relx, rely):
    self.x=self.x+relx
    self.y=self.y+rely

  "Draw Method"
  def draw(self):
    print "("+str(self.x)+
          ","+str(self.y)+")"

point=Point(1,2)
point.draw()    # (1,2)

# Modifies attributes of a
single object
point.z=3
print point.z       # 3

# Modifies methods of a
single object
def draw3D(self):
  print "("+str(self.x)+
        ","+str(self.y)+
        ","+str(self.z)+")"

point.draw3D=draw3D
point.draw3D()      # (1,2,3)

# Modifies methods of a class
def getX(self):
  return self.x

Point.getX=getX
print point.getX()  # 1

# Modifies attributes of
# every Point instance
Point.isShowing=0
```

a) Python Source Code

```ruby
class Point
  def initialize(x, y) # Constructor
    @x=x
    @y=y
  end
  def move(relx, rely) # Move method
    @x+=relx
    @y+=rely
  end
  def draw() # Draw method
    puts "("+String(@x)+","+String(@y)+")"
  end
end

point=Point.new(1,2)
point.draw() # (1,2)

# Modifies attributes of a single object
def point.z=(value)
  @z=value
end
point.z=3

# Modifies methods of a single object
def point.draw3D()
  puts "("+String(@x)+","+String(@y)+
       ","+String(@z)+")"
end
point.draw3D() # (1,2,3)

# Modifies methods of a class
class Point
  def getX()
    return @x
  end
end
puts point.getX() # 1

# Modifies attributes of a class
def Point.isShowing=(value)
  @@isShowing=value
end
Point.isShowing=false
```

b) Ruby Source Code

Fig. 1. Python and Ruby examples of structural reflection.

to benefit from all the advantages mentioned above. The main reason why we have selected the .NET abstract machine was its design focused on supporting a wide number of languages [38,39]. The approach of using a free JIT compilation framework, such as OpenJIT [40], would involve a lower runtime performance and limitations on reusing existing libraries, frameworks and tools. Conversely, extending the .NET platform to support dynamically typed languages facilitates future interoperation with existing languages and any .NET application or component.

Once we had decided to use the .NET abstract machine, our next decision was the selection of an implementation (a specific virtual machine). As we have explained in the motivation section, we need a shared source implementation to extend its semantics and an efficient JIT compiler to improve runtime performance of reflective primitives. The SSCLI (Shared Source Common Language Infrastructure) implementation of the Microsoft .NET platform has been our

choice because, although there exist other .NET platform implementations (such as Mono [41] or DotGNU Portable.NET [42]), the SSCLI it is nearer to the commercial virtual machine implementation: the Common Language Runtime (CLR).

Microsoft SSCLI, also known as Rotor, is a source code distribution that includes fully functional implementations of the ECMA-334 C# language standard and the ECMA-335 Common Language Infrastructure specification, various tools, and a set of libraries suitable for research purposes [43]. The SSCLI runs on Windows XP, FreeBSD 4.5 and Mac OS X.

SSCLI consists of 3.6 million lines of code that can be divided into 4 groups:

(1) **The Execution Environment**. This is the virtual machine of the .NET platform that includes the JIT compiler, a generational garbage collector, the class loaders, and the Common Type System. The source code of the execution environment, commonly called IL (Intermediate Language), is encapsulated in managed executables.
(2) **The Libraries**. The SSCLI distribution includes the source code of its Base Class Library (BCL), runtime reflection (structural introspection), XML processing, and extended array classes. There are also additional libraries included in this distribution, most notably the support for regular expressions and an extensive framework for type serialization, object remoting, and automatic type marshalling. The BCL provides types to represent the built-in data types of the CLI (Common Language Infrastructure), simple file access, custom and security attributes, string manipulation, formatting, streams, collections, and so forth. All these services are included in the `System` namespace.
(3) **Compilers and tools**. SSCLI includes compilers for C# (ECMA-334) and JScript entirely written in C#. It also consists of a set of tools such as a managed code debugger, an assembler, a disassembler, an assembly linker, and a stand-alone verification tool.
(4) **Platform Abstraction Layer (PAL)**. This code implements the abstraction layer between the runtime environment and the operating system. The PAL exposes a collection of 242 interfaces that must be implemented on each target platform.

In this project we have extended the execution environment to adapt the semantics of the abstract machine, obtaining a new reflective computational model that is backward compatible with existing programs –see the next section. We have also extended the Base Class Library (BCL) to add new structural reflection primitives, instead of defining new IL statements. We refer to this new platform as Reflective Rotor or ЯRotor

4 Design

As we have mentioned, runtime reflective features of the SSCLI are restricted to the introspection level: system structure can be dynamically consulted but not modified. At the same time, the .NET platform offers the facility to dynamically generate IL code at runtime in a limited way (it only permits to create new types, not modifying existing ones) by means of its `System.Reflection.Emit` namespace.

In our project, we have extended the introspective capabilities of .NET CLI at the abstract machine level, adding the set of structural reflective primitives offered by many dynamically typed languages. A new namespace has been added to the BCL: `System.Reflection.Structural`. We will show in Section 5.1 which are its specific services, but its functionality can be grouped into:

- **Field manipulation**. Besides modifying the structure of a class (altering the structure of all its instances), we can also alter the composition of a single object (object-level reflection). Fields may be added, deleted or replaced.
- **Method manipulation**. Methods of classes can be dynamically added, replaced and erased. Therefore, the set of messages accepted by an object could change at runtime depending on its dynamic context. This dynamic typing scheme is also known as "duck typing": *if it walks like a duck and quacks like a duck, it must be a duck* [2]. It means that an object is interchangeable with any other object that implements the same dynamic interface, regardless of whether the objects have a related inheritance hierarchy or not.

  A new method could also be placed in a single object (object-level reflection). The body of these new methods can be obtained from existing ones, or dynamically generated by means of the `System.Reflection.Emit` namespace.

The programmer could combine these facilities with the introspective services that .NET already offers, making the CLI an appropriate platform to develop language-neutral adaptive software.

4.1 Class-Based Object-Oriented Model

There exist some conceptual inconsistencies between the class-based object-oriented computational model and structural reflection. These inconsistencies were first noticed and partially solved in the field of object-oriented database management systems [44]. In this area, objects are stored but their structure,

or even their types (classes), could be altered afterwards as a result of software evolution.

The first scenario of modifying the structure of a class implies updating the structure of all its instances. This mechanism has been defined as a type reclassification operation in class-based languages, meaning that it is possible to change the class membership of an object while preserving its identity [34,45]. It has also been referred to as schema evolution in the database world. The modification of class instances could be performed as soon as the class is evolved (eager) or when the object is up to be used (lazy) [46]; it is only necessary to know the type (class) of an object at runtime. Dynamic evolution of class methods and fields can produce situations such as accessing fields or methods that do not exist in a specific execution point; these situations are detected by a dynamic type checking mechanism, in order to make sure that no incorrect behavior is produced. This is how Smalltalk provides dynamic modification of classes.

There is another situation that a structurally reflective computational model should support, but in this case is much more difficult to model it in a class-based language. How can the structure of an object be modified without altering the rest of its class instances? This problem was detected in the development of MetaXa, a reflective Java platform implementation [31]. The approach they chose was also adopted by some object-oriented database management systems: schema versioning [47]. A new version of the class (called "shadow" class in MetaXa) is created whenever an object is reflectively modified. This new class is the type of the recently customized object.

The schema versioning approach causes different problems such as maintaining the class data consistency, class identity, using class objects in the code, garbage collection, inheritance reliability, and memory consumption, involving a really complex and difficult to manage implementation [48]. One of the conclusions of the MetaXa research project was that the class-based object-oriented model does not fit well in structural reflective environments. They finally stated that *the prototype-based model would express reflective features better than class-based ones* [48].

4.2 Prototype-Based Object-Oriented Model

In the prototype-based object-oriented computational model the main abstraction is the object, suppressing the existence of classes [49]. Although this computational model is simpler than the one based on classes, there is no loss of expressiveness; i.e. any class-based program can be translated into the prototype-based model [50]. A common translation from the class-based

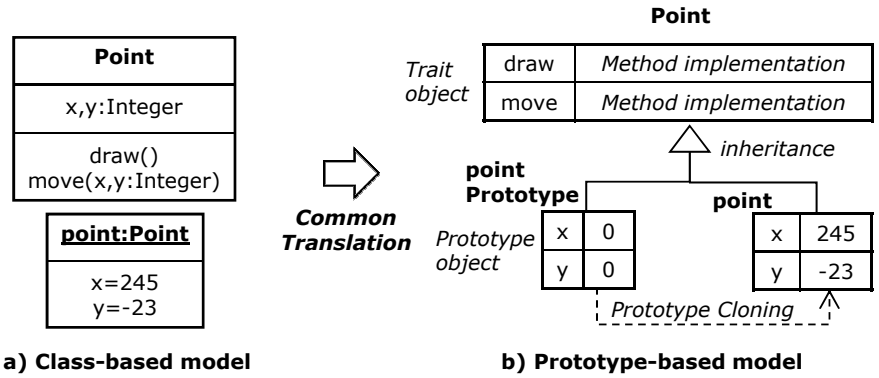

Fig. 2. Common translation scheme between class-based and prototype-based object models.

object-oriented model is the one shown in Figure 2:

- Similar object behavior (methods of each class in the class-based model) can be represented by trait objects: objects whose members are only methods [51]. Thus, their derived objects share the behavior they define.
- Similar object structure (fields of each class in the class-based model) can be represented by prototype objects. A prototype object holds a set of initialized fields that represent a common structure.
- Copying prototype objects (constructor invocation in the class-based model) is equivalent to the creation of a new class instance. A new object with a specific structure and behavior is created by cloning a prototype object.

There exist class-based languages (Java, Smalltalk or C#) where classes are first class objects represented by objects at runtime (e.g., in the .NET platform, instances of `System.Type` are objects that represent classes or another type). This demonstrates that, without any loss of expressiveness, this translation is intuitive and facilitates application interoperability. This is the reason why this model has been considered as a universal substrate for object-oriented languages [52,53].

For our project, the most important feature of the prototype-based object-oriented computational model is that it models structural reflective primitives in a consistent way. Structural reflective languages such as Moostrap [54], Self [55] or Lua [4] have successfully employed this model. The prototype-based object model overcomes the schema versioning problem stated in the previous section [56]. Modifying the structure (fields and methods) of a single object can be performed directly, because any object maintains its own structure and even its specialized behavior. Since shared behavior is placed in trait objects, their customization implies type adaptation (schema evolution).

Figure 3 shows the example scenario described in the source code of Figure 1. The initial `point` and `p2` objects are clones of the `pointPrototype` object and their shared behavior is placed in the `Point` trait object. A new coordinate

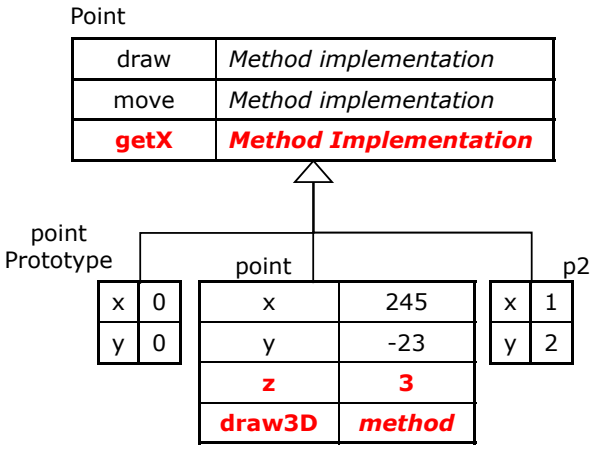

Fig. 3. Structural Reflection over the prototype-based object-oriented model.

field (`z`) has only been added to `point`. Using the same approach, only the `point` object is able to `draw3D` its three coordinates. Finally, all the objects derived from the `Point` trait object will be able to use the new `getX` method.

Adding a field to a trait object represents a new variable shared by all of the instances. As we have seen in the motivating example (Figure 1) Ruby follows this approach, whereas Python implements a schema evolution technique where all class instances should be involved. We will show how we have implemented both approaches in Section 5.

Although most structural-reflective dynamically-typed languages provide classes, the concept of class many of them use (e.g., Python, Ruby and JavaScript) is not exactly the same as the one used by other class-based languages such C++, Java or C#. Classes in the former group of languages do not represent both shared behavior and structure of objects. Structures of objects can be modified at runtime (object-level reflection) without changing their classes. That is, classes simply model shared behavior, the same as trait objects in the prototype-based computational model. Objects are responsible for storing their own structure, and they can also contain specific behavior (methods) —like in the prototype-based computational model.

4.3 The New Virtual Machine Computational Model

Object-oriented dynamically typed languages that offer object-level structural reflection use the prototype-based model to implement structural reflection in an appropriate way. However, although the so called *Common Language Infrastructure* (CLI) tries to support a wide set of languages, the .NET platform only offers a class-based object-oriented model optimized to execute statically typed languages. For example, Visual Basic .NET and Boo [57] are two of programming languages that offer class-based dynamic (and static) typing

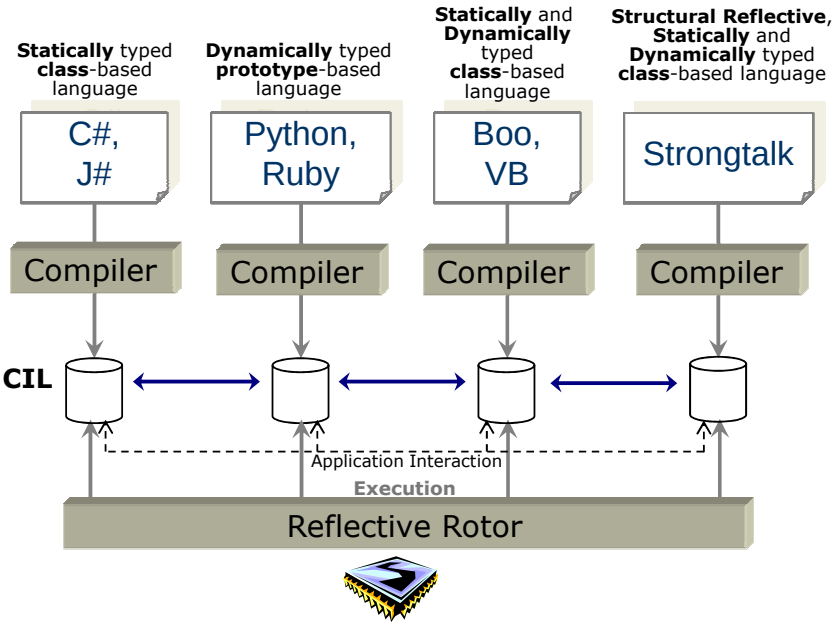

Fig. 4. Supporting different object-oriented languages' models.

features. They use the introspective services of the .NET platform in order to implement dynamic typing, but this approach implies a significant performance detriment at runtime.

In order to make prototype-based dynamically typed languages interoperable with existing .NET languages and applications, the class-based model should be maintained, ensuring backward compatibility. The reflective virtual machine should be able to run any existing .Net application, producing the same original behavior.

Taking into account that the CLI virtual machine is a low level platform for executing a wide variety of high-level programming languages, our work is aimed at supporting both class-based and prototype-based object-oriented models: the former for running static class-based .NET applications; the latter for executing dynamic reflective programs. Each .NET compiler could then select services of the appropriate model, depending on the language being compiled. Figure 4 shows this scheme.

(1) In our reflective version of Rotor (ЯROTOR), it is possible to execute any existing .NET language (e.g., C# or J#) application, previously compiled for the original SSCLI. Since we have just extended its computational model, our reflective version of the SSCLI is backward compatible. Existing programs will use the original class-based model of the CLI.
(2) To support different dynamically typed languages that offer structural reflection (e.g., Python or Ruby), we have added structural reflection primitives with prototype-based semantics. Consequently, a compiler of any dynamically typed language will be able to directly use these new

services of the reflective platform. It is not necessary to generate extra code for simulating a reflective model over a static one.

(3) There are also programming languages compiled to .NET that use both dynamic and static typing. Two examples are Visual Basic for .NET and Boo [57]. These languages do not support structural intercession; its dynamic type system only supports introspection. Dynamic features of these languages will benefit from the performance improvement granted by our reflective virtual machine.

(4) Strongtalk is a Smalltalk modification where static typing is optional, introducing type-checking in a reflective prototype-based object-oriented language without compromising flexibility [58]. It supports runtime structural reflection at the class level —it does not permit the modification of a single object. A Strongtalk compiler may statically infer types or postpone type-checking to runtime, using both capabilities of our virtual machine.

In conclusion, the objective of our reflective extension of the .NET platform is supporting structural reflection with both class and prototype based computational models, implementing static and dynamic typing (performed by the compiler or the virtual machine, respectively). Future work will include defining the interaction between different models that are executed over the same runtime environment –see the conclusions and acknowledgments sections.

As we have previously mentioned, we have maintained the use of classes to achieve both backward compatibility and class-based structural reflection. Therefore, the structural manipulation of classes could involve the following scenarios:

- **Schema evolution for class-based languages**. Since classes are first class objects in the .NET platform, their structure is customized by means of `System.Type` instances. Altering their methods produces adaptation of object behavior. In case we adapt fields of `System.Type` objects (classes), what we obtain is the customization of all the existing instances of the adapted class. Looking for a good runtime performance, we have developed a lazy schema evolution mechanism [46]. This adaptation of classes has been parameterized with a boolean argument indicating whether the new member is an instance or class field (`static` in C++, Java and C#). Dynamic typing detects the use of non-existing members, throwing the appropriate runtime exception if necessary —see Section 5.1.
- **Traits customization in the prototype-based model**. Modifying the interface of a trait object implies the customization of object shared behavior (the same semantics as described above). However, the meaning of class structure modification (its fields) depends on the language. As we have seen in our example, Python interprets this modification as the adaptation of all the existing instances of the class being customized (lazy schema evolu-

tion); in Ruby, however, it simply implies the manipulation of class (static) members. Both functionalities are included in the paragraph above.

Finally, the scenario of modifying the structure of objects is only applicable to dynamic prototype-based languages. This operation is meaningless in class-based languages because classes define the invariant behavior and structure of all their instances. However, in a reflective prototype-based model it is possible to customize the specific behavior (methods) of a single object and its structure (fields). These operations do not need to adjust the structure of classes because they only represent shared behavior (trait objects), overcoming the schema versioning problem described above. Note that compilers of statically type-checked .NET languages (e.g., C#) will never use these object-oriented reflective features because of its statically typed class-based model.

5 Implementation

Implementing structural reflective services in the SSCLI requires the extension of the CLI platform, involving two discussions:

(1) **Where to place the structural reflection primitives**. New reflective services could either be added to the Base Class Library (BCL), or represented by new IL statements. Adding new reflective IL statements would provide two benefits: the JIT-compiler would have more opportunities for optimization, and the virtual machine loader could detect some semantic errors prior to program execution.

On the other hand, placing the reflection services inside the BCL also provides some advantages. With the library-centric approach, a transparent use of reflection would be offered to every .NET programming language, improving the reusability of the reflective services [59]. It is not necessary to modify existing compilers to use the reflective library. Future implementations might even reuse the runtime if they are implemented with bytecode transformation at load-time, like Javassist [60] and Kava [61]. Finally, the most important advantage of maintaining the ECMA-335 standard is that existing *Commercial Off-the-Shelf* (COTS) .NET software could be reflectively manipulated. COTS programs are deployed in the binary IL format. With the BCL-based approach, it would be possible to load third-party binary code into memory and reflectively manipulate its classes and objects, reflecting the changes on its execution. Therefore, we finally decided to place the reflective primitives in the BCL, maintaining the existing IL instruction set. These new services have been added to the Base Class Library (BCL) in a new namespace called `System.Reflection.Structural`.

(2) **Execution of third-party components**. Using third-party compo-

nents in reflective scenarios can be performed in two different ways. The first one is extending the semantics of the platform, maintaining the original components intact. The other approach is adapting the ECMA-335 to a new reflective specification, translating existing binary components to our new platform.

The adaptation of the ECMA-335 would be done refining the type system of the CLI to represent this dynamism [62], including, for instance, a static mechanism for supporting duck typing similar to structural types in Scala [63] or a constraint-based type system [64]. At the same time, aliasing would be another issue to tackle. Modifying the structure of an object would imply the static modification of its structure. All the references that may be pointing to the modified object should be identified to statically type-check its new structure. This problem can be addressed by implementing an alias analysis algorithm [65].

The alternative we have chosen is extending the semantics of IL, because it implies, from our point of view, an important benefit: it maintains the portability of existing executable COTS components taken from third-parties. This approach preserves the Common Type System (CTS, ECMA-335 Partition I) and the Common IL instruction set (CIL, ECMA-335 Partition III). Therefore, our selected approach has been to enhance (but not to modify) the semantics of some IL statements, to represent the new reflective prototype-based model. As an example, the inheritance strategy commonly used in static languages is based on concatenation, whereas some dynamic languages implement a delegation-based inheritance strategy [66]. In order to obtain this behavior, we have extended the semantics of the `call` and `callvirt` IL statements, maintaining backward compatibility with existing applications.

5.1 New Reflective Primitives

We present a summary of the most significant reflective primitives added to the `System.Reflection.Structural` namespace (all of them are `static` methods of the `NativeStructural` utility class). The semantics of these services are the one described in Section 4:

- The {`add`, `remove`, `alter`, `get`, `exist`}`Method` methods receive an object or class (`System.Type`) as the first parameter, indicating whether we want to modify (or inspect) a single object or a shared behavior. The second parameter is a `MethodInfo` object of the `System.Reflection` namespace. This object uniquely describes the identifier, parameters, return type, attributes and modifiers of a method. The `IsStatic` property of the `MethodInfo` instance is used to select the schema evolution behavior (prototype-based language) or class member adaptation (class-based language).

If the programmer needs to create a new method, he or she can generate it by means of the `System.Reflection.Emit` namespace, and add it later to an object or class using its `MethodInfo` instance.

- The `invoke` primitive executes the method of an object or class specifying its name, return type and parameters. If no reflection has been used, a fast concatenation strategy is used. However, in the execution of reflective dynamically typed languages, method invocation is based on delegation: when a message is passed to an object, it is checked whether the object has a suitable method or not; in case it exists, it is executed; otherwise, the message is passed to its class (its trait object) recursively. A `MissingMethodException` is thrown if the message has not been implemented in the hierarchy.
- The {`add`, `remove`, `alter`, `get`, `exist`}`Field` methods modify the runtime structure of single objects (prototype-based model) or their common schema (classes or traits) passed as the first parameter. The second parameter is an instance of a new `RuntimeStructuralFieldInfo` class (derived from the .NET `FieldInfo` class) that describes the type of the field, its visibility, and its attributes. Once again, the `Static` attribute of the second parameter selects the schema evolution behavior (class-based and Python models) or class member adaptation (class-based and Ruby semantics).

The code in Figure 5 is a partial C# translation of the Python reflective program shown in Figure 1 that uses the new reflective primitives. The addition of the new `isShowing` field to the `Point` class has been done following both the Python and Ruby style.

The functionality of these BCL services has been implemented in the C programming language inside the execution environment. The most important implementation decision was finding the place to add the reflective information of each object and class. Whenever an object is created in the heap, it holds two pointers followed by the instance data (values of fields). The first on is a `MethodTable` pointer that holds the memory address of the type method-table, which follows a concatenation inheritance strategy [66]. The second pointer points to the object's syncblock: a 32-bit integer index into a cache of SyncBlocks. The syncblock memory contains additional control structures of each object such as thread synchronization locks or value types [43].

Since instance data of each object is placed after the method table pointer and the syncblock index, our first approach was trying to enhance this instance data at runtime. However, direct object structure manipulation turned to be much more problematic than we expected. Since the SSCLI internals has been designed to aggressively optimize the support of static class-based languages, objects and classes have fixed-size data and a very tight memory layout. This fact made us pursue a different implementation path: we stored additional members into each object's syncblock, which was designed to hold additional control structures of each object or class. The syncblock is stored

```
public class Point {
    ... // * Fields and methods of the Point class

    public static void Main() {
        Point point = new Point(1, 2);
        point.draw();
        // * Modifies fields of a single object
        RuntimeStructuralFieldInfo rsfi = new RuntimeStructuralFieldInfo("z",
                  typeof(int), 3, FieldAttributes.Public);
        NativeStructural.addField(point, rsfi);
        Console.WriteLine(NativeStructural.getValue(point, "z"));
        // * Modifies methods of a single object
        MethodInfo draw3D = typeof(Point3D).GetMethod("draw3D");
        NativeStructural.addMethod(point, draw3D);
        NativeStructural.invoke(point, draw3D, draw3D.GetParameters(), null);
        // * Modifies methods of a class
        MethodInfo getX = typeof(Point3D).GetMethod("getX");
        NativeStructural.addMethod(typeof(Point), getX);
        Console.WriteLine(NativeStructural.invoke(point, getX,
                  getX.GetParameters(), null));
        // * Modifies fields of a class
        rsfi = new RuntimeStructuralFieldInfo( "isShowing",
                  typeof(bool),false, FieldAttributes.Public);
        NativeStructural.addField(typeof(Point), rsfi); // the Python way
        rsfi.SetAttributes(rsfi.Attributes | FieldAttributes.Static);
        NativeStructural.addField(typeof(Point), rsfi); // the Ruby way
    }

// * The following code could also be generated at runtime
//   with Refleciton.Emit
class Point3D {
    public void draw3D() {
        Console.WriteLine("({0},{1},{2})", this.x, this.y, this.z);
    }
    public int getX() {
        return this.x;
    }
}
```

Fig. 5. C# version of the motivating example.

into the private execution engine memory rather than in the heap managed by the garbage collector [43]. Every object in the system could have an associated syncblock. If an object is removed from memory, the runtime environment automatically deletes it [43]. This behavior provides a way to adequately manage data stored in the syncblock.

The syncblock size can be safely modified, so we added specific structures to store reflective information (Figure 6). These new structures are two hash tables (dynamically added methods and fields are stored separately) that map member identifiers to member handles. The syncblock of every class and object in the SSCLI follows a lazy creation strategy. No syncblock is created if it is not necessary, involving no memory consumption overhead. In case the SSCLI

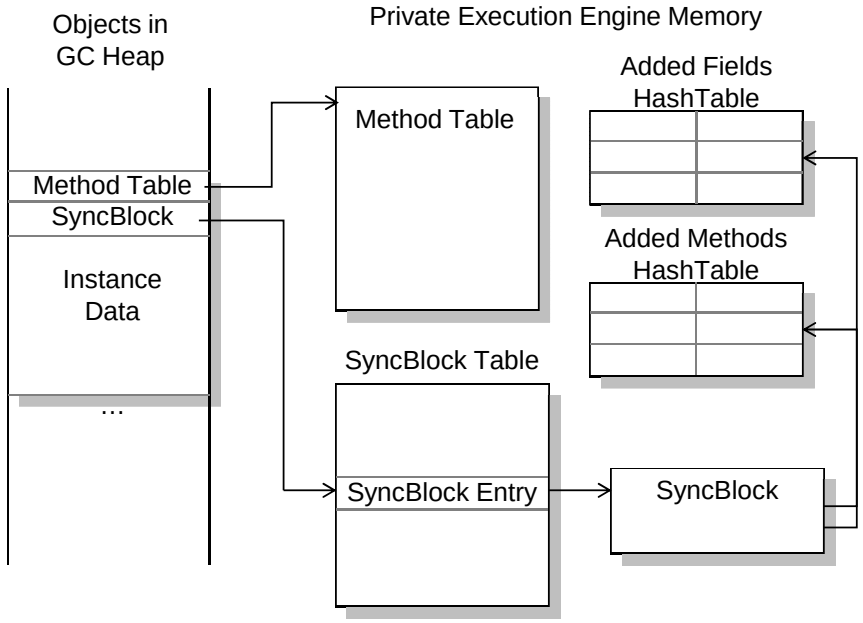

Fig. 6. Placing the reflective information inside the SyncBlock.

requires a syncblock for non-reflective purposes (e.g., thread synchronization or saving value types), the space overhead introduced by our approach is limited to a pair of null pointers.

This memory representation is valid for modeling both concatenation and delegation inheritance semantics. If a statically typed class-based language is being processed, the information is obtained from the original SSCLI structure. In case it is a prototype-based reflective language, information is also consulted in the object's syncblock following a delegation strategy [66].

The inclusion of structural reflective information into the syncblock made us tackle with the following key issues:

(1) **Representation of dynamic fields**. Information of dynamically created fields must be accurately represented. For that, we have added the `RuntimeStructuralFieldInfo` class to the `System.Reflection.Structural` namespace. This class inherits from the `RuntimeFieldInfo` standard BCL class, used to store field meta-information. This new class stores the data needed to handle dynamic fields, making it compatible with static ones (that use `RuntimeFieldInfo` instead). As a result, the system could use both types indistinctly.
(2) **Representation of dynamic methods**. Classes and objects are able to store new methods. These methods can be either created at runtime, or copied from existing ones. New methods can be dynamically created with the `System.Reflection.Emit` namespace. Since copying existing methods to create new ones may imply a performance cost, we have used method wrappers to simulate this copy. When an existing method

obtained from an object or class is copied to another one, an instance of the `MethodWrap` class is created, holding a pointer to the original method. When a `MethodWrap` is executed, it delegates the invocation to the method it is pointing to.

(3) **Method invocation**. Depending on the computational model of the language, method invocation should be implemented using a concatenation strategy or a delegation-based one. If no structural reflection service has been used, we maintain the concatenation-based message passing mechanism implemented by the SSSLI [43]: the syncblock is not analyzed and the SSCLI `MethodTable` (*vtable*) is directly used. In case structural reflection has been used, we follow the delegation approach instead. The search of the method to be executed starts by analyzing the object's syncblock. If the method is not found, its actual type (class) is checked at runtime. The syncblock of the class (reflective member set) is analyzed first, and then its `MethodTable` (original member set). If the method is still not found, this algorithm is recursively applied to its superclass. Finally, in case no superclass exists, a `MissingMethodException` is thrown. With this scheme, both computational models are supported. At the same time, dynamic binding semantics is not changed because we start searching at the actual type of the object.

(4) **Alteration and deletion of members**. Since data and method tables of instances have fixed sizes, member handles in the syncblock are also responsible for managing the deletion of non-reflective members — implementation details are described in the following section. When a method is deleted, we do not physically erase it; it is only marked as deleted. This way, it is possible to maintain the execution of a method that is on the call stack while it is being removed. If the method is then called, a `MissingMethodException` will be thrown. Method alteration is actually performed using the deletion and addition operations.

(5) **Interaction with garbage collector (GC) operations**. The reflective information is stored into the execution engine memory (the syncblock), which is not part of the GC heap. However, dynamically created fields and methods refer to memory placed in the GC heap that may be moved or deleted by the GC. Fortunately, the SSCLI manages a data structure (the handle table) that allows data stored into the execution engine memory to point to data placed in the GC heap [43]. This way, the hash tables added to the syncblock store member handles pointing to entries into a handle table, holding the actual member data. Since we decided to use GC data structures offered by the SSCLI, the main consequence expected of our enhancement is a performance cost of scanning a higher amount of members (those added at runtime), and the consequent memory consumption –evaluated in Section 6.

(6) **Concurrency**. The fact that methods code is not physically removed is particularly important when different threads that use structural reflection are created. Since the SSCLI does not implement a *code pitching*

```
 // * Static typing message passing
ldloc point
callvirt instance void Point::draw()
 // * Dynamic typing message passing
ldloc point
callvirt instance void System.Object::draw3D()
```

Fig. 7. Part of the IL version of the motivating example.

mechanism (act of releasing native code from the JIT heap), the binary code of the method is actually never freed. This is one of the reasons why structural reflection could cause important memory consumption in the SSCLI —see Section 6.2. Regarding to synchronization, the handle table not only supports GC interaction, but also thread synchronization. The `CreateHandle` function is used when members are about to be added, and the `DeleteHandle` one is used for deleting them. These two functions are synchronized to manage handles in a thread-safe way, supporting the concurrent manipulation of fields.

5.2 Extending the Semantics of the Abstract Machine

The use of the new reflective BCL services should involve the adaptation of running programs. However, legacy non-reflective code does not make explicit calls to the BCL `Reflection.Structural` namespace and, thus, reflective changes will not be taken into account within the original program. For instance, a third-party binary application uses the `callvirt` IL statement to pass a message to an object, instead of our BCL `invoke` primitive. This is the reason why the reflective model defined in Section 4 requires extending the semantics of some specific IL statements, making existing COTS applications adaptable. What we achieve with this approach is making existing .NET components adaptable without needing to recompile them.

Following our motivating example, Figure 7 shows how to invoke a method following both the static and dynamic typing approaches. We first send a `draw` message of the statically inferred type `Point`. In case the compiler does not know the type of the object, a dynamically typed message call is needed (`draw3D`); then, the `Object` (or any other) type should be used instead. What we have done to produce this behavior is extend the semantics of the `callvirt` IL statement with the dynamically typed computational model described in this paper.

In order to achieve this goal, we have modified the native code the JIT compiler generates for the following IL statements:

- `ldfld`, `ldsfld` and `ldflda`: Loads the (instance or class) field value (or address) into the stack following the computational model described in Section 4.3.

- `stfld` and `stsfld`: Stores a value into a (instance or class) field, allocating its appropriate memory location at runtime.
- `call` and `callvirt`: Executes a method following both the concatenation and delegation inheritance strategies.

The semantics of these IL statements have been extended to allow dynamic access to its reflective information, not available when the code is compiled. This has been accomplished with two major actions:

(1) Modifying the assembly code that the JIT compiler generates at runtime, when the above statements are about to be executed: functions `compileCEE_{LDFLD, LDFLDA, STFLD, CALL, CALLVIRT}`. The original JIT compiler chooses between two methods to generate member accesses: 1) generating assembly code that makes direct access to object members, whose memory addresses are statically calculated; or 2) generating code that performs calls to special SSCLI helper functions that access system data at runtime, looking for the appropriate member. In the original JIT compiler, 1) is the preferred method leaving 2) to handle some special cases. We have moved a hefty amount of 1) cases to the 2) category, enabling member accesses to explore the reflective information added at runtime through calls to helper functions. It has also been required to move some existing JIT static type-checking code to these helper functions. It is necessary to avoid JIT compilation errors and perform type checking at runtime, which is when member types are actually known.

(2) Modifying the previously mentioned helper functions that are called at runtime by the generated assembly code: `JIT_{Set, Get}Field{32, 64, Obj}`, `JIT_GetFieldAddr`, `JIT_GetStaticFieldAddr` and `JIT_Test{Method, VirtualMethod}` functions. Two new helper functions were also added in order to obtain a method call address: `JIT_Test{Method, VirtualMethod}`. These helpers take into account both the "static" member information (the instance data and its method table) and the new reflective information placed in the syncblock. The result is an extension of the original SSCLI computational model, offering the behavior defined in Section 4. Reflective primitives placed in the `NativeStructural` class also make use of these helper functions, and hence both ways of accessing the structural reflective services share the same internal code.

An important aspect of the described JIT modifications is that the IL instruction set remains unchanged. It is the assembly code that the JIT compiler generates for each IL instruction what handles all the changes. When the JIT compiler is about to generate native code that accesses an object member, the kind of member and the type of access are analyzed to determine which helper function is more suitable. A call to a helper is generated instead of a direct access to the member. The returned value of this helper function is then used by the native code as the offset of the member.

In order to implement deletion of non-reflective members, information in the syncblock prevails over the original member information of the SSCLI. In our motivating example, if the `x` field of a point is removed, it cannot be actually erased from the "static" instance data. To solve this, a deletion mark is added to the dynamic information of the object (into its syncblock). The preference of the reflective information will represent the elimination of the `x` field.

5.3 Generics and Structural Reflection

We have developed our reflective platform extending the SSCLI version 1.0. Since generics has been included in the last version of the SSCLI (version 2.0), this section describes the major issues that should be addressed to incorporate generics to our current implementation.

The main issue is how the SSCLI 2.0 manages generic types. Types are represented by `TypeHandle`s that may point to either a `TypeDesc`, or a new `TypeVarTypeDesc` object that represents a type variable. Each class has a `MethodTable` that collects a set of methods, described by `MethodDesc` objects. A generic non-instantiated class represents its generic types (type variables) by means of `TypeHandle`s that point to `TypeVarTypeDesc` objects. Whenever a generic class is instantiated specifying its actual types, a new `MethodTable` object is created substituting its generic types with the concrete ones (`TypeHandle`s now point to `TypeDesc` objects). An `ExposedClassObject` instance in the new `MethodTable` represents the class instance at runtime, and its syncblock would be used to hold the reflective data of the instantiated class. Therefore, a generic class has a representation for its non-instantiated version, plus as many representations as existing instantiations of its type variables. Structural reflection may be applied to each one of these different representations of a class.

If a non-instantiated generic class is modified with reflection, every instantiated version of that class should also be modified; transitively, every object should be altered as well. These changes could be reflected following the lazy strategy described in Section 5.1. Since the SSSLI represents an instantiated generic class as a non-generic one, its modification would not require any special consideration. Adding methods or fields that use generic types should be avoided when the class is not a non-instantiated generic class.

The SSCLI 2.0 also supports generic methods. Generic methods could be placed in (and hence reflectively added to) any class. Types of generic methods are also represented with `TypeVarTypeDesc` objects, but type instantiation is much simpler. In this case, the compiler is the one that infers types at each method call. Consequently, the `call` and `callvirt` IL instructions explicitly

state actual types of parameters and return values. A dictionary holding substitutions of each generic type is used at method execution to replace type variables with the corresponding actual types.

The BCL in SSCLI 2.0 has been extended with a set of classes and methods that support generic types via reflection. The `GenericTypeParameterBuilder` class represents a type variable. Considering that this class inherits from `System.Type`, it could be used to represent generic types without changing the interface of our reflective API.

6 Evaluation

This section presents detailed experimental results showing the effectiveness of our work. The experimental methodology employed is firstly outlined. Afterwards, the benchmark classification and applications used for the evaluation are discussed. Finally, we present a performance and memory consumption assessment.

6.1 Methodology

Different benchmarks have been used to assess the efficiency of our implementation, measuring both runtime performance and memory consumption. Three different sets of tests have been run to evaluate:

- **Efficiency of structural reflection**. A set of micro-benchmarks and a real reflective application have been used to measure efficiency of programs that make extensive use of the reflective features of dynamically typed languages.
- **Non-reflective code**. We have measured runtime performance and memory consumption of code that does not use reflection at all. Different benchmarks have been used to evaluate efficiency of different dynamically typed languages implementations.
- **The cost of reflection**. The original implementation of the SSCLI 1.0 for Windows on *free* mode (its fastest version, enabling optimizations and disabling debug code and debugging symbols) has been compared with our reflective platform. We have used real applications that do not employ any of the new features added to the SSCLI described in this paper. In this section we also compare the results with the CLR 1.1 build 1.1.4322 CLI implementation.

In order to compare our reflective SSCLI 1.0 implementation with existing dynamically typed languages, we have selected both Python and Ruby program-

ming languages because of their wide popularity and utilization at present. The specific implementations of Python and Ruby we have used are:

- **CPython 2.5.1 for Windows** (commonly referred as simply Python). This is the most widely used Python implementation; it is called CPython because it has been developed in C.
- **Jython 2.2** (formerly called JPython) over the Java HotSpot Client VM build 1.6.0.01 for Windows. A 100% pure Java implementation of the Python programming language. It is seamlessly integrated with the Java 2 platform.
- **IronPython 1.1** over the CLR 2.0 build 2.0.50727 for 32 bits. A promising implementation of the Python language targeting the Common Language Runtime (CLR). It compiles Python programs into IL bytecodes that run on either Microsoft's .NET or the Mono open source platform.
- **Ruby 1.8.6 for Windows**. Ruby is a dynamic open-source programming language with a focus on simplicity and productivity. Ruby has probably become a popular programming language because of the success of the Ruby on Rails Web development framework [9]. Its "official" implementation is based on a C interpreter.

These implementations have been compared to ЯRotor, our extension of the SSCLI 1.0 for Windows. The system has been compiled in the *free* operation mode, without debug information and with the highest degree of code optimization.

The source code we have used to assess ЯRotor is the CLI intermediate language (IL). In order to obtain the IL code, we first write high-level programs in C# and then compile them down to IL. Since IL code has static type annotations, we replace them with `Object` references —as the source code shown in Figure 7. The resulting code not only measures structural reflection, but also dynamic type checking. Therefore, we measure the same operations in our platform and each implementation of the Python and Ruby programming languages.

The code has been instrumented with hooks to evaluate runtime performance, recording the value of the processor's time stamp counter. We have measured the difference between the value between the beginning and the end of each benchmark to obtain the total execution time of each program. This assessment method takes into consideration the time required to dynamically generate native code by the JIT compiler of the virtual machine.

All the benchmarks have been executed utilizing the Windows XP performance monitor. We have measured the maximum size of working set memory used by the process since it started (the `PeakWorkingSet` property). The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an appli-

cation to use without triggering a page fault. The working set includes both shared and private data. The shared data comprises the pages that contain all the instructions that the process executes, including those from the process modules and the system libraries.

All tests have been carried out on a lightly loaded 3.2 GHz iPIV hyper-threading system with 1 GB of RAM running Windows XP. To evaluate average percentages, ratios and orders of magnitude, we use the geometric mean.

6.2 Structural Reflective Code

In order to evaluate the efficiency of the structural reflective primitives added to the SSCLI, we have implemented (in Ruby and Python) a micro-benchmark that makes extensive use of all the reflective primitives described in this paper (loops of 10,000 iterations). This assessment gives us an initial estimate of runtime performance improvement and memory consumption cost of our implementation, in comparison with the implementations described in Section 6.1. We also evaluate the execution of a real program that makes use of structural reflection.

Table 1: Performance and memory consumption of reflective primitives.

Reflective Primitive	CPython	Jython	IronPython	Ruby	ЯRotor
1. Adding int fields to an object	439 ms 4,700 KB	18,240 ms 34,781 KB	8,461 ms 24,839 KB	219 ms 9,560 KB	47 ms 6,028 KB
2. Adding object fields to an object	486 ms 6,726 KB	18,137 ms 34,984 KB	10,277 ms 24,847 KB	263 ms 9,236 KB	31 ms 5,948 KB
3. Adding int fields to a class	530 ms 4,894 KB	17,611 ms 34,970 KB	8,279 ms 24,963 KB	686 ms 9,976 KB	47 ms 6,884 KB
4. Adding object fields to a class	485 ms 5,894 KB	17,378 ms 35,225 KB	10,052 ms 25,671 KB	657 ms 9,960 KB	47 ms 5,952 KB
5. Adding int fields to a class an access to its value	688 ms 4,906 KB	32,716 ms 34,887 KB	16,032 ms 24,259 KB	829 ms 9,948 KB	109 ms 6,863 KB
6. Adding object fields to a class an access to its value	845 ms 5,916 KB	31,392 ms 35,602 KB	17,651 ms 24,514 KB	799 ms 9,948 KB	109 ms 6,622 KB
7. Deleting int fields from an object	391 ms 4,704 KB	16,526 ms 34,024 KB	7,865 ms 24,689 KB	204 ms 9,344 KB	109 ms 6,288 KB
8. Deleting object fields from an object	421 ms 5,702 KB	16,929 ms 35,124 KB	7,914 ms 25,362 KB	188 ms 9,612 KB	125 ms 6,270 KB
9. Deleting int fields from a class	451 ms 4,900 KB	16,737 ms 34,980 KB	7,984 ms 24,323 KB	421 ms 10,092 KB	125 ms 6,504 KB
10. Deleting object fields from a class	435 ms 5,894 KB	16,932 ms 34,778 KB	8,167 ms 25,482 KB	406 ms 10,112 KB	125 ms 6,270 KB

Table 1: Performance and memory consumption of reflective primitives.

Reflective Primitive	CPython	Jython	IronPython	Ruby	ЯRotor
11. Accessing fields from an object	455 ms	16,959 ms	7,907 ms	219 ms	16 ms
	4,820 KB	35,264 KB	24,347 KB	12,780 KB	5,590 KB
12. Accessing fields from a class	421 ms	16,954 ms	7,963 ms	219 ms	16 ms
	5,016 KB	35,136 KB	25,121 KB	12,780 KB	5,792 KB
13. Accessing added fields from an object	455 ms	16,959 ms	7,907 ms	187 ms	94 ms
	4,820 KB	35,264 KB	24,347 KB	10,164 KB	6,284 KB
14. Accessing added fields from a class	421 ms	16,954 ms	7,963 ms	187 ms	78 ms
	5,016 KB	35,136 KB	25,121 KB	10,576 KB	6,284 KB
15. Accessing non-existing fields from an object	499 ms	18,604 ms	9,856 ms	593 ms	375 ms
	4,240 KB	34,839 KB	24,311 KB	10,164 KB	5,202 KB
16. Accessing non-existing fields from a class	469 ms	17,900 ms	9,926 ms	593 ms	391 ms
	4,244 KB	34,537 KB	24,421 KB	10,576 KB	5,206 KB
17. Adding methods to an object	110 ms	5,361 ms	2,369 ms	125 ms	187 ms
	4,156 KB	33,281 KB	23,437 KB	8,680 KB	18,670 KB
18. Adding methods to a class	125 ms	4,863 ms	2,425 ms	109 ms	187 ms
	4,224 KB	31,929 KB	23,971 KB	8,676 KB	18,670 KB
19. Invoking methods that were added to an object using reflection	141 ms	4,723 ms	2,671 ms	79 ms	63 ms
	4,160 KB	32,702 KB	23,714 KB	8,652 KB	19,612 KB
20. Invoking methods coded statically	141 ms	5,581 ms	2,671 ms	63 ms	16 ms
	8,406 KB	35,391 KB	52,709 KB	9,232 KB	21,100 KB
21. Invoking non-existing methods	157 ms	5,684 ms	3,197 ms	141 ms	94 ms
	4,096 KB	30,675 KB	23,653 KB	9,904 KB	5,644 KB
22. Invoking methods that were added to a class using reflection	141 ms	4,914 ms	2,671 ms	63 ms	63 ms
	4,210 KB	32,652 KB	24,081 KB	9,176 KB	19,610 KB
23. Deleting methods that were added to an object	125 ms	4,521 ms	1,970 ms	110 ms	31 ms
	4,158 KB	32,910 KB	23,834 KB	8,606 KB	18,920 KB
24. Deleting methods that were added to a class	111 ms	4,457 ms	1,991 ms	78 ms	31 ms
	4,214 KB	32,770 KB	23,980 KB	9,310KB	18,920 KB

Table 1 shows the measurement of each primitive execution time expressed in milliseconds and the Kbytes needed to execute them (following the methodology described in Section 6.1). As we can appreciate in this table, Jython and IronPython obtain the worst performance results in all the tests. The requirement to implement Jython as a 100% pure Java offers interoperability with any Java program, but it causes a significant performance penalty. The same happens to IronPython: generating IL code that simulates the Python reflective model over a platform that does not support it involves low performance at runtime. This performance penalty is surely caused by the amount of extra code that must be generated to support the reflective model.

Data in Table 1 shows that CPython, Ruby and our implementation execute

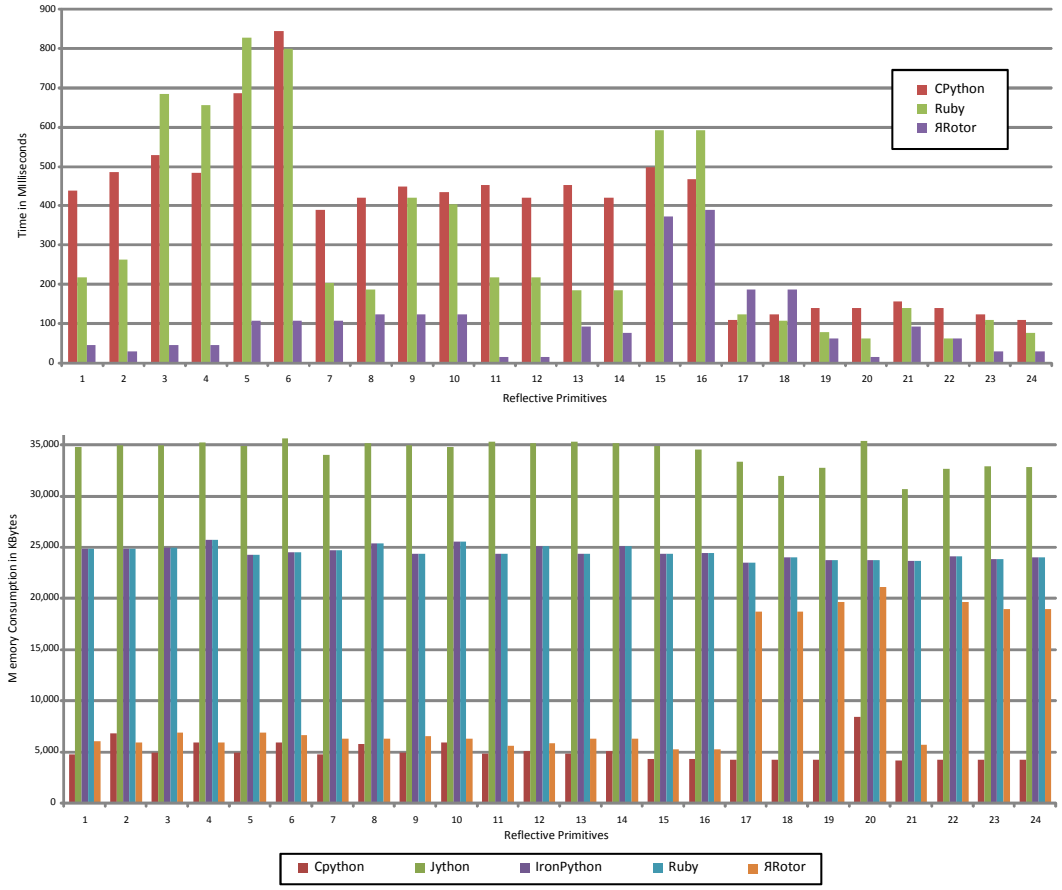

Fig. 8. Runtime performance and memory consumption of reflective primitives.

structural reflective primitives much faster than the two systems that generate intermediate code: Jython and IronPython. Measurements also show that our platform is more than 80 and 160 times faster on average than IronPython and Jython respectively. Note that, since time values of Jython and IronPython are much higher than the rest of implementations, we have decided not to include them in Figure 8, displaying only the systems whose performance is close enough to be compared. Regarding to memory consumption, Jython requires more than 3 times the memory used by ЯRotor and the memory needed by IronPython is almost twice higher than ours.

As Figure 8 shows, when running the reflection test suite with the new reflective prototype-based semantics added to the SSCLI runtime environment, we are 3.317 times faster than CPython and 2.135 times faster than Ruby. This performance improvement of ЯRotor compared to CPython involves a memory consumption increase of 73.2%. However, our implementation utilizes 86.67% the memory employed by Ruby.

The only tests where Reflective Rotor has been slightly slower are those where methods are added at runtime (primitives 17 and 18). This difference might be caused by the way SSCLI manages methods. An object handle is the only

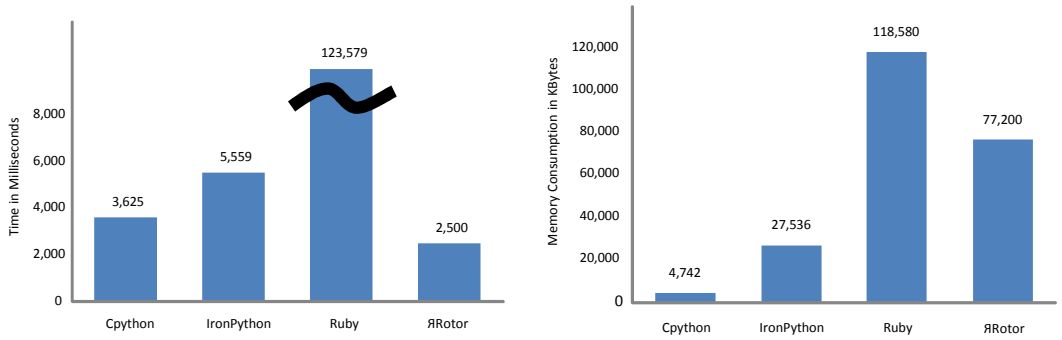

Fig. 9. Runtime performance and memory consumption of the parrot benchmark.

way to access the members of an object (or class). This indirection is the mechanism used to implement references managed by a generational garbage collector. Regarding to methods, it is necessary to build their mangled name, obtain their object handle, and access to their implementation. The cost of this whole process might be the reason of the observed performance drop.

The three primitives that offer a lower benefit are those where the code tries to access non-existing members (primitives 15, 16 and 21). This cost is due to the complexity of the member searching algorithm (delegation strategy) necessary in the dynamic reflective prototype-based model. This is a drawback produced by the design of the CLI, which was built to simply support statically typed programming languages.

These reflective micro-benchmarks illustrate a first performance assessment of structural reflection. However, the impact of these results on the whole performance of real applications depends on the amount of reflective code used. Although dynamically typed languages are commonly chosen by their flexibility, real applications that make use of these services must be evaluated to estimate its overall performance.

Since we measure structural reflection in this section, we will evaluate a real program benchmark that uses reflection. More real workloads are analyzed in sections 6.3 and 6.4. The benchmark we have used is the Parrot benchmark 1.0.4. This benchmark was created to measure the corners of the Python language, using the dynamic object model of this language to implement a Python interpreter. It implements a parser for a subset of Python, instrumenting and uninstrumenting the tree visiting algorithms at runtime, making use of common meta-programming services of dynamically typed languages. Runtime performance and memory consumption are shown in Figure 9. Jython has not been included because it does not support the `yield` statement, used many times in the source code of the test.

The inclusion of code that does not use reflection produces different results that will be even more obvious in the following tests. We can see how ЯRotor

is significantly faster than CPython and IronPython (45% and 122.36% respectively) and much faster (more than 48 times) than Ruby. In this case, our reflective platform is the fastest one.

Analyzing the memory used to run the parrot benchmark, it becomes clear that, excluding Ruby, the approaches that use a JIT-compiler virtual machine require considerably more memory than the interpreter-based ones. IronPython and ЯRotor use 4.8 and 15.28 times more memory than CPython. The difference between IronPython and ЯRotor could be because of the *code pitching* mechanism (act of releasing native code from the JIT heap) implemented in the CLR, which has not been included in the SSCLI implementation. Since the benchmark generates a lot of new code and replaces existing methods at runtime, the CLR dynamically releases the compiled code that becomes inaccessible, whereas the SSCLI does not.

6.3 Non-Reflective Code

The main advantage of so called dynamic languages is their capabilities (such as reflection) to model dynamically adaptive and adaptable software. Therefore, we thought that it is important to evaluate the efficiency of their dynamic features (previous point). However, there are pieces of "static" code when developing an application in a dynamically typed language. Therefore, this section is focused on assessing the efficiency of "static" programs that do not make use of reflection at all. Note that, under these circumstances, statically typed languages (C#, Java or C++) would be more appropriate to develop this kind of software.

In this section we have used two benchmarks. The first one was designed by Thomas Bruckschlegel to evaluate the characteristics of Java, C#, and C++ on Windows and Linux. This benchmark comprises a set of 14 elementary tests that use fundamental data processing and arithmetic operations [67]. We have translated the benchmark source code into Python and Ruby.

The second benchmark we have used is the Pystone benchmark. This benchmark is the Python version of the Dhrystone benchmark [68] and is commonly used to compare different implementations of the Python programming language. Pystone is included in the standard CPython distribution. We have translated it into Ruby and C#.

Table 2 shows the results of executing both benchmarks in the languages described in Section 6.1. CPU time is expressed in milliseconds and memory consumption in Kbytes.

Table 2: Performance and memory consumption of non-reflective benchmarks.

Test	CPython	Jython	IronPython	Ruby	ЯRotor
1. Integer Arithmetic	20,641 ms	31,719 ms	10,480 ms	69,578 ms	1,500 ms
	4,266 KB	15,118 KB	22,716 KB	6,852 KB	4,220 KB
2. Double Arithmetic	27,516 ms	37,813 ms	14,980 ms	94,359 ms	5,750 ms
	4,266 KB	13,838 KB	21,323 KB	8,000 KB	4,640 KB
3. Long Arithmetic	45,625 ms	98,406 ms	33,558 ms	410,532 ms	3,781 ms
	4,266 KB	14,476 KB	21,418 KB	8,262 KB	4,664 KB
4. Trigonometric	5,203 ms	14,453 ms	4,052 ms	4,453 ms	453 ms
	4,278 KB	14,608 KB	23,980 KB	8,332 KB	4,668 KB
5. File Input / Output	32 ms	277 ms	203 ms	93 ms	312 ms
	4,344 KB	14,828 KB	27,576 KB	8,332 KB	4,668 KB
6. Array set and get operations	6,000 ms	41,484 ms	3,990 ms	19,157 ms	359 ms
	4,406 KB	15,072 KB	27,576 KB	10,788 KB	4,668 KB
7. Exception Handling	859 ms	3,250 ms	10,255 ms	3,125 ms	1,812 ms
	4,220 KB	15,118 KB	23,252 KB	9,151 KB	5,968 KB
8. Hashmap fill and find operations	31 ms	360 ms	420 ms	125 ms	31 ms
	6,858 KB	16,894 KB	27,994 KB	10,362 KB	5,968 KB
9. Nested Hashmaps find, get, and set operations	62 ms	343 ms	217 ms	343 ms	203 ms
	6,858 KB	17,616 KB	27,994 KB	10,362 KB	5,968 KB
10. Heap Sort Algorithm	2,407 ms	13,532 ms	1,731 ms	6,094 ms	281 ms
	5,564 KB	17,604 KB	31,074 KB	12,452 KB	7,328 KB
11. Double-Linked Lists add, get and remove operations	46 ms	250 ms	70 ms	78 ms	31 ms
	5,564 KB	19,144 KB	31,074 KB	12,452 KB	7,328 KB
12. Matrix Multiply operation.	132,782 ms	347,610 ms	83,650 ms	569,719 ms	16,328 ms
	5,564 KB	18,425 KB	26,149 KB	15,396 KB	8,128 KB
13. Nested Loops performing add operations.	34,060 ms	58,328 ms	25,341 ms	318,438 ms	1,468 ms
	5,564 KB	18,587 KB	22,297 KB	12,163 KB	8,132 KB
14. String Concatenation.	8,703 ms	39,164 ms	6,092 ms	30,461 ms	359 ms
	16,944 KB	52,043 KB	57,694 KB	41,916 KB	23,952 KB
Pystone benchmark	1,069 ms	3,829 ms	764 ms	3,173 ms	156 ms
	4,373 KB	12,912 KB	23,616 KB	8,208 KB	6,004 KB

Table 2 shows how Reflective Rotor executes non-reflective code significantly faster than the rest of implementations. On average, ЯRotor is 2.95, 13.83, 4.36 and 5.06 times faster than CPython, Jython, IronPython and Ruby respectively. Although ЯRotor requires more memory than CPython and Python, these differences are lower than the performance benefit of our implementation: CPython uses 49.58% the memory required by Reflective Rotor. Jython, IronPython and Ruby increase the memory consumption of ЯRotor in 58.87%, 147.36% and 71.63%.

Figure 10 shows the ratios of execution time to CPython and the ratios of

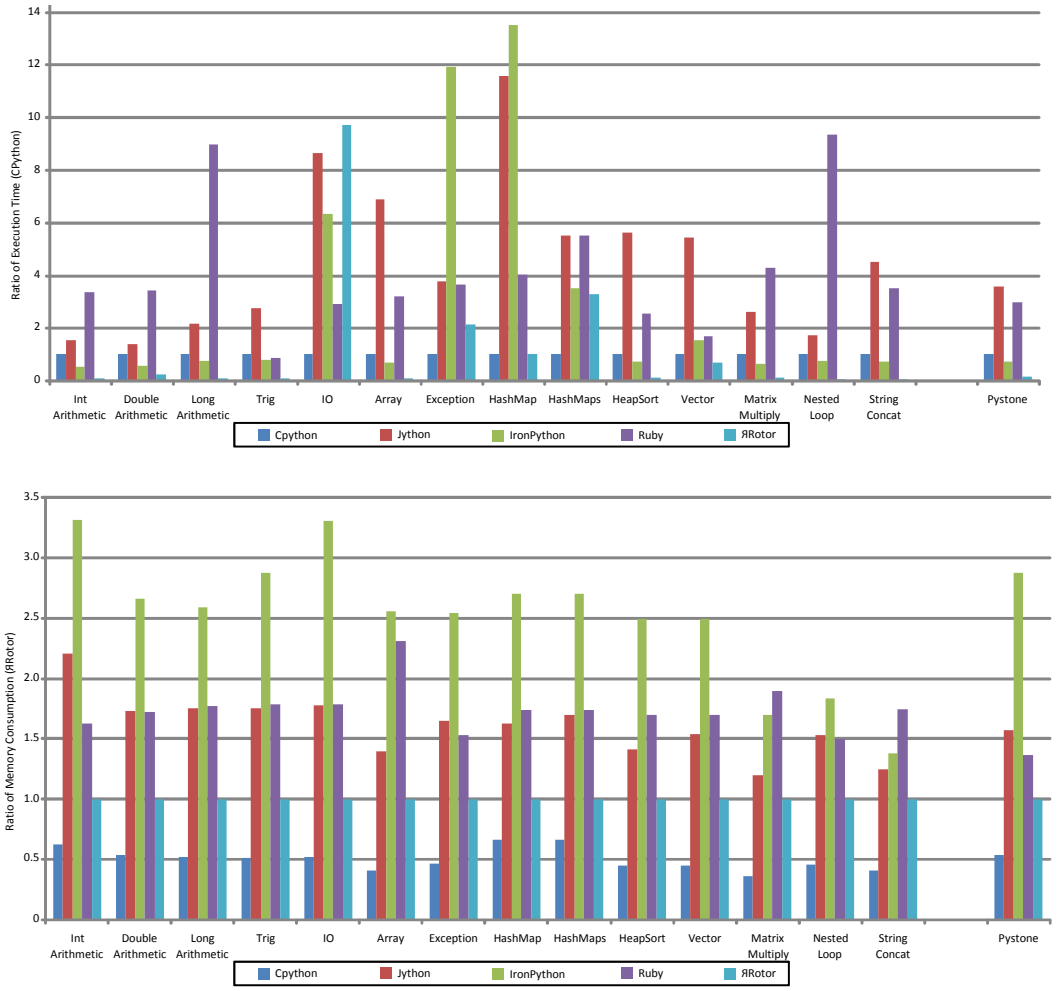

Fig. 10. Non-Reflective performance and memory consumption.

memory consumption to ЯRotor. Table 2 and Figure 10 illustrate how tests were CPython is faster than ЯRotor are the same as those where CPython is also faster than IronPython: file input/output, exception handling, hash map fill and find operations, nested hash maps find, get, and set operations, and matrix multiply operation of the Bruckschlegel benchmark. That coincidence implies that both the CLR (IronPython) and the SSCLI (ЯRotor) perform part of these tests slower than the Python interpreter. We think this difference is because the key elements of these tests (hash-tables, vectors and files) are implemented inside of the .Net library, whereas they are part of the interpreter in the case of Python. BCL access may cause this performance penalty. Regarding to exception management, both the CLR and SSCLI have poor performance in their implementations of the exception handling mechanism [67].

On average, our extension of the SSCLI executes these tests that do not use reflection 2.53 times faster than CPython, using 101.69% more memory. In most cases (10 tests from 15), IronPython is also faster than CPython. Al-

though IronPython uses the CLR implementation of the CLI, which is faster than the SSCLI, IronPython is not as fast as ЯRotor. This dissimilarity is caused by the additional layer that IronPython implements over the CLR to simulate the whole computational model of Python. IronPython has to determine types of variables at runtime using introspection. Unlike ЯRotor, the CLR shows an important performance penalty when introspection is used.

As we have previously mentioned, most implementations of dynamically typed language runtimes are slower than statically typed ones when programs do not need runtime adaptation. This is mainly caused by the fact that dynamically typed languages discover the types of objects at runtime, whereas this process is performed at compile-time by statically typed ones. Therefore, when a programmer chooses a dynamically typed language to implement a program, it is most likely motivated by the flexibility requirements of the application, which are best offered by dynamically typed languages. This is the reason why we believe it is more important to measure performance of reflective primitives rather than non-reflective code. Good performance results obtained in both scenarios implies that our reflective extension of the SSCLI is a really appropriate platform to support hybrid scenarios.

6.4 The Cost of Reflection

This last evaluation section compares the SSCLI implementation with our platform, using the same programming language (C#). We ran a set of benchmarks that do not use reflection at all. The results will give us an estimate of the cost of our reflective model when class-based static applications are executed. We also compare our base system (the SSCLI) with the production CLR. This assessment estimates what might be the efficiency of our system in case it was included in the CLR implementation.

We have measured runtime performance and memory consumption of three C# benchmarks: Thomas Bruckschlegel benchmark used in previous section [67]; three real C# applications collected by Ben Zorn [69]; and a C# port of a subset of the Java Grande benchmark [70].

The three C# real applications collected by Ben Zorn consist of a collection of managed code benchmarks available for performance studies of CLI implementations. These programs are:

- **LCSCBench**. Based on the front end of a C# compiler. It uses a generalized LR (GLR) parsing algorithm. This benchmark is compute and memory intensive, requiring hundreds of megabytes of heap for the largest input file provided (a C# source file with 125,000 lines of code).
- **AHCBench**. Based on compressing and uncompressing input files using

Adaptive Huffman Compression. AHC bench is 1,267 lines of code compute-intensive, requiring a relatively small heap.

- **SharpSATBench**. Based on a clause-based satisfiability solver where the logic formula is written in Conjunctive Normal Form (CNF). SharpSAT-bench is compute-intensive, requiring a moderate-sized heap. Its source code has 10,900 lines of code.

Table 3: Performance and memory consumption costs of reflection.

	Test	ЯRotor		SSCLI		CLR	
		CPU Time	Memory	CPU Time	Memory	CPU Time	Memory
Zorn	LCSCBench	3,906 ms	29,729 KB	3,484 ms	26,896 KB	1,859 ms	34,764 KB
	AHCBench	5,672 ms	5,807 KB	5,359 ms	5,375 KB	859 ms	5,320 KB
	SharpSATBench	6,469 ms	10,888 KB	4,310 ms	10,358 KB	1,340 ms	14,942 KB
Java Grande	Arith	5,188 ms	4,296 KB	5,188 ms	4,036 KB	4,530 ms	3,986 KB
	Assign	1,134 ms	4,316 KB	469 ms	4,308 KB	31 ms	4,212 KB
	Cast	859 ms	4,564 KB	859 ms	4,244 KB	422 ms	4,212 KB
	Create	22,625 ms	6,498 KB	22,625 ms	6,300 KB	5,641 ms	4,216 KB
	Loop	531 ms	4,556 KB	531 ms	4,232 KB	31 ms	4,148 KB
	FFT	32,844 ms	37,247 KB	31,594 ms	36,994 KB	12,609 ms	37,196 KB
	HeapSort	4,094 ms	8,182 KB	2,984 ms	7,924 KB	578 ms	6,158 KB
	Sparse	13,516 ms	8,992 KB	13,078 ms	8,726 KB	5,406 ms	9,042 KB
	RayTracer	131,422 ms	5,218 KB	90,063 ms	5,016 KB	4,938 ms	4,806 KB
Bruckschlegel	Integer Arithmetic	1,500 ms	4,220 KB	1,500 ms	3,988 KB	593 ms	4,276 KB
	Double Arithmetic	5,750 ms	4,640 KB	5,750 ms	4,312 KB	1,093 ms	4,288 KB
	Long Arithmetic	3,781 ms	4,664 KB	3,781 ms	4,336 KB	1,531 ms	4,288 KB
	Trigonometric	453 ms	4,668 KB	453 ms	4,336 KB	250 ms	5,496 KB
	File Input / Output	312 ms	4,668 KB	312 ms	4,336 KB	46 ms	5,496 KB
	Array	359 ms	4,668 KB	359 ms	5,464 KB	15 ms	5,496 KB
	Exception Handling	1,812 ms	5,968 KB	1,703 ms	5,492 KB	1,093 ms	5,568 KB
	Hashmap	31 ms	5,968 KB	31 ms	6,464 KB	15 ms	5,568 KB
	Hashmaps	203 ms	5,968 KB	156 ms	6,464 KB	46 ms	5,568 KB
	Heap Sort	281 ms	7,328 KB	265 ms	6,464 KB	31 ms	5,568 KB
	Vector	31 ms	7,328 KB	31 ms	6,464 KB	15 ms	5,568 KB
	Matrix Multiply	16,328 ms	8,128 KB	16,328 ms	7,648 KB	2,359 ms	8,988 KB
	Nested Loops	1,468 ms	8,132 KB	1,468 ms	7,648 KB	453 ms	9,000 KB
	String Concatenation	359 ms	23,952 KB	250 ms	23,468 KB	46 ms	17,936 KB

The last benchmark used in this section is a subset of the Java Grande benchmark ported to C# by Chandra Krintz [71]:

- **Section 1 (low level operations)**. *Arith*, execution of arithmetic operations; *Assign*, variable, object and class variables, and array assignment; *Cast*, casting between different primitive types; *Create*, object and array creation; and *Loop*, loop overheads.
- **Section 2 (Kernels)**. *FFT*, one-dimensional forward transformation of N complex numbers; *Heapsort*, the heap sort algorithm over arrays of integers; and *Sparse*, management of an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure.
- **Section 3 (Large Scale Applications)**. *RayTracer*, a 3D ray tracer of scenes that contain 64 spheres, and are rendered at a resolution of NxN pixels.

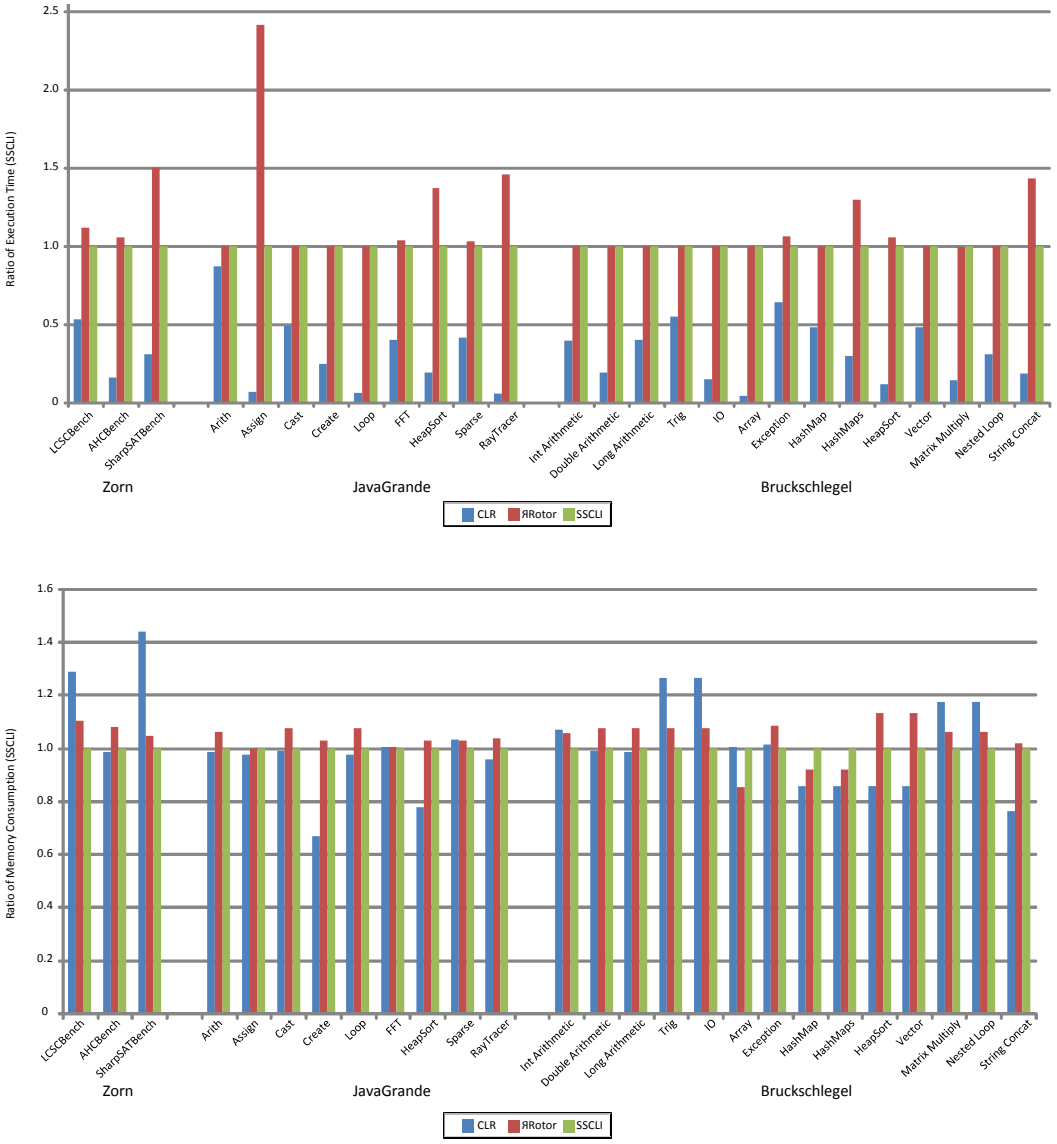

Fig. 11. Performance and memory cost of runtime reflection.

Table 3 presents the results; time is expressed in milliseconds and memory in Kbytes. Figure 11 shows execution time and memory consumption ratios to the SSCLI. Both representations show how our implementation involves an average runtime performance cost of 12.10%, and 4.27% more memory utilization than the SSCLI. If we compare runtime performance of the SSCLI with the CLR, the latter is 3.15 times faster than the former, using only 38% more memory.

Although memory consumption variance is low (6.32%), the standard deviation of runtime performance penalty is 30.32%. This is because a large number of tests have almost no performance cost, whereas others show a performance penalty. This difference could be clearly seen in the two heap sort algorithm implementations: although the cost is only 3% in the case of the JavaGrande

benchmark, this percentage raises to 13% in the case of the Bruckschlegel benchmark. Analyzing the code, we realized that the main difference is that the former sorts an array local variable when the latter uses an object field. This result was contrasted with the rest of the tests. Programs where a notable performance penalty is observed are those that make more accesses to object members. Consequently, we evaluated member access and method invocation costs with a simple micro-benchmark. The results obtained converged to the following percentages:

- The runtime performance cost of accessing an object's field is 31.65%. This value is 165.53% in the case of static (class) fields. The higher performance penalty of static field access is due to the worse performance of the `JIT_GetStaticFieldAddr` helper function compared to the `JIT_GetFieldAddr` –see Section 5.2.
- Method invocation involves a performance penalty of 28.29% when the message is sent to an object and 27.06% when the receiver is a class.

This assessment confirms performance penalties shown in Figure 11. The assign benchmark evaluates different kind of assignments, making a wide use of static fields (its performance penalty is 142%). The rest of the tests where the cost is appreciable have penalties lower than 50%.

7 Related Work

There have been different approaches to speed up implementations of dynamically typed object-oriented reflective languages using JIT compilation and dynamically adaptive code optimization. However, not many have tried to do it supporting both dynamically and statically typed programming languages.

7.1 Runtime Reflective Virtual Machines

The Smalltalk programming language could be identified as the first example of a dynamically typed object-oriented reflective language. It supports class-level intercession, but not object-level intercession. This is the reason why its computational object-oriented model is class-based. Initial implementations of Smalltalk (Dolphin, GNU Smalltalk, ObjectStudio or Berkeley Smalltalk) were based on bytecode interpreters. Afterwards, different optimizations have used dynamic JIT compilation to native code [7] involving important performance improvements: VisualWorks, VisualAge Smalltalk and Digitalk. As an example, average runtime performance of VisualWorks Smalltalk is more than 3 times better than GNU Smalltalk.

Self is a prototype-based object-oriented language for exploratory programming. It uses the prototype model to support runtime structural (and partially behavioral) reflection. Its implementation is based on a virtual machine that implements JIT compilation [55]. One of the most important features of Self is the efficient execution of its dynamic code [26]. The Self compiler transparently specializes functions for specific argument types based on profiling and gatherer statistics. To implement a consistent reflective object model, the Self platform was designed following a prototype-based approach.

MetaXa, formerly called MetaJava, is an extension of the Java platform with a reflective meta-level architecture [31]. Reflective services are provided by a behavioral reflective Meta-Object Protocol (MOP) [32]. The MetaXa approach is quite similar to the one presented in this paper: reflection support added to a production statically typed class-based virtual machine (integrated into its JIT compiler) to obtain significant performance benefits [72]. The main difference was that MetaXa followed the class-based computational model of the Java programming language. As described in Section 4, the class-based object-oriented model of Java does not support object-level reflection in a consistent way [48]. In fact, this model is not the one implemented by most dynamically typed object-oriented reflective languages.

7.2 Python Implementations

Python is a dynamically typed reflective object-oriented programming language used in many software development scenarios. Python has also been ported to the Java and .NET platforms. Probably, the most widely used implementation is Cpython, a free and efficient bytecode interpreter written in C.

ActiveState tried to modify different free implementations of the .NET platform in order to compile Python to .NET IL code, but they abandoned the project because *the abstract machine design was not friendly to dynamic languages* [27]. They built some prototypes, but all of them had poor performance.

Python for .NET is an open source Zope Public License (ZPL) implementation that extends CPython with a package that gives programmers nearly seamless integration with the .NET Common Language Runtime (CLR). This package does not implement Python as a first-class CLR language (it does not produce managed code from Python code). Rather, it is an integration of the CPython engine with the .NET runtime. This makes its implementation another version of CPython.

Jython is an implementation of Python seamlessly integrated with the Java platform. The predecessor to Jython, JPython, is certified as 100% pure Java.

Concerning to performance, CPython 2.3 on Windows 2000 is about twice as fast as Jython 2.1 on JDK 1.4. Due to its lower performance, Jython is commonly used as a scripting language embedded in Java.

IronPython is a shared source platform implementation that supports the Python interactive programming style with dynamic JIT compilation. IronPython was created by Jim Hugunin (the creator of JPython/Jython) under a Common Public License (CPL) until version 0.6. Then, Jim Hugunin has worked in the IronPython project as a Microsoft employee, releasing IronPython 0.7 to 1.0 as a BSD-style license (MS Shared Source Initiative). IronPython 1.1 executes the Pystone benchmark 1.8 times faster than CPython 2.5. This benefit is because of the "static" code of the Pystone benchmark. However, the *dynamic field lookup and replacing type* test of the Parrot benchmark runs 65 times slower over Mono and 1.5 times slower over the CLR 2.0. IronPython has demonstrated how the JIT compiler of the CLR offers an important performance benefit when executing code that does not use dynamic features of Python. However, the simulation of its dynamic services over the statically typed .NET platform causes a performance penalty in dynamic scenarios compared to CPython.

7.3 Ruby

Ruby is a dynamic pure object-oriented programming language. It is nowadays getting widely popular, probably due to the success of the Ruby on Rails framework for developing database-backed web applications [9]. Using the runtime reflection and meta-programming features of Ruby, Rails notably facilitates the development of Web-based programs.

Runtime performance of Ruby is poor because it is not compiled to a virtual machine, and it uses neither native threads nor generational garbage collection. This is the reason why future implementation of Ruby 2 interpreter, called Rite, will be a bytecode-based virtual machine to improve its runtime performance [73].

Another approach to speed-up the execution of Ruby programs is the Cardinal project. This open source project is intended to compile Ruby programs so that they can be run over the Parrot JIT-based virtual machine. Cardinal is still in version 0.1.0. Parrot virtual machine last version is 0.9.0.

The increasing popularity of dynamically typed languages has produced different approaches to make the Java Virtual Machine a platform to execute dynamic languages. The Java `instrument` package (included in Java SE 1.5) provides services that allow Java programming language agents to instrument programs running on the JVM. This package has been used to implement JAsCo, a fast dynamic AOP platform [74]. Other tools like BCEL [75] and Javassist [60] have been successfully used in the implementation of application servers like Spring Java and JBoss, obtaining good runtime performance.

Afterwards, the JSR 223 was included as part of the Java SE 1.6. It provides an API to access scripting language programs developed in the Java Platform, permitting the use of scripting language pages in Java server-side applications [11].

Currently, the JSR 292 is another step forward to support dynamically typed languages on the Java platform. JSR 292 is expected to be delivered in Java SE 1.7 [11]. The JSR 292 extends the JVM instruction set with a new `invokedynamic` opcode. This instruction has been designed to support the implementation of the message passing mechanism of dynamically-typed object-oriented languages (*duck typing*). The specification also investigates support for *hotswapping*: the capability to modify the structure of classes at runtime.

Since the computational model of dynamically typed languages requires extending the JVM semantics, Sun Microsystems has launched the new Da Vinci project in January 2008 [76]. This project is aimed at prototyping a number of enhancements to the JVM, so that it can run non-Java languages, specially dynamic ones, with a performance level comparable to that of Java itself. Although capabilities of Da Vinci Machine are planned for inclusion in the upcoming Java SE 7, Sun has not provided a release date for JDK 7 and it is not known how many Da Vinci features might be actually included in JDK 7 [77].

The last approach taken by Sun Microsystems is similar to the one presented in this paper. Instead of creating a new software layer (like IronPython or Jython), they extend the virtual machine to take advantage of its JIT compiler. At the same time, they will support both dynamically and statically typed languages. However, the new `invokedynamic` instruction only supports the *duck typing* feature of dynamic languages. No structural reflective services have been described yet.

7.5 Dynamic Language Runtime (DLR)

Microsoft first announced in the MIX 07 the Dynamic Language Runtime (DLR): a set of services that run on the top of the CLR, offering a new level of support for dynamic languages on .NET [78]. The DLR is shipped with IronPython 2.0 beta 4 and SilverLight 2 beta 2.

Basically, the DLR is a redesign of the object model used in previous versions of IronPython. The DLR has been developed to facilitate the implementation of dynamically typed languages over .NET and to make these languages seamlessly work together, sharing libraries and frameworks. Its services have been used to implement IronRuby, IronPython 2.0, dynamic Visual Basic .NET and Managed JScript. Since the reflective code is executed over the CLR, the performance of dynamic reflective code is similar to the previous versions of IronPython.

7.6 Optimizations at the High Level

The execution of dynamically typed languages (and reflective features of other statically typed languages like Java or C#) can also be optimized at a higher level of abstraction. The following approaches implement performance optimizations, using features of high-level programming languages.

Psyco is a *just-in-time specializing compiler* that runs existing Python software faster, with almost no change in the source code [79]. Taking advantage of the Python meta-programming features, Psyco gathers information of programs at runtime, writes several optimized versions of each function, and executes them properly. This runtime program translation is called *specialization-by-need*: specialization is performed dynamically, when the code is about to be executed. Psyco does not implement a JIT compiler of a virtual machine. It follows the premise that *high-level languages need not be slower than low-level ones* [80]. Therefore, its approach is based on just-in-time high-level code specialization. Although this approach is not the one presented in this paper, both techniques can be combined to obtain an even better runtime performance. Program specialization optimizes a specific high-level programming language. A reflective JIT-compiler virtual machine improves the runtime performance of a language-neutral platform. Notice that just-in-time program specialization requires runtime information of objects structure and dynamic method invocation, and the runtime performance of these features is significantly better in our reflective platform than in the original SSCLI.

PyPy is a concluded European project aimed at producing a fast Python implementation, translating a Python-level description of the Python language

itself to lower level languages. PyPy is based on a framework that supports the generation of implementations of dynamic languages, separating language specification and implementation aspects [81]. This separation improves flexibility of implementation decisions such as platform, memory or optimizations. PyPy architecture is based on an interpreter of Python written in a subset of Python called *RPython*. A translation tool compiles RPython programs into efficient lower-level programs for various target platforms, including JIT-compiler virtual machines. This high-level JIT compilation obtains good runtime performance but depends on the Python high-level programming language, lacking the language interoperation feature offered by language-neutral virtual machines.

The *SmartReflection* project consists on optimizing the Java core reflection library [82]. The main idea of this approach consists of moving the most of the overhead of Java dynamic introspection from runtime to compile-time. A static bytecode processor creates stub classes that resolve the reflective method overloading statically, and delegates the real invocation to the standard mechanism (not the reflective one). At runtime, method reification is modified using the Java Native Interface (JNI) to dynamically locate appropriate stub classes and transform reflective calls into direct method invocations. The resulting speed up of the introspective `invoke` primitive is up to 60%.

8 Conclusion

This paper describes how to modify the computational model of an efficient statically typed class-based JIT-compiler virtual machine in order to support structural object-oriented reflection at runtime, obtaining a significant runtime performance improvement. Moreover, computational models of both prototype-based and class-based programming languages are implemented. Therefore, existing .NET programming languages are still supported by our virtual machine.

Previous implementations that support dynamically-typed object-oriented languages over a .NET and Java virtual machines (e.g., Jython and IronPython) have obtained no performance improvement when running structural reflective code. Due to the non-reflective object-oriented model of the virtual machines used, these compilers use an abstraction layer that simulates dynamic reflective features over these statically typed platforms. This new abstraction layer requires the execution of extra code, causing a runtime performance penalty. When non-reflective code is executed, these platforms obtain better performance than their interpreter-based counterparts. However, in the case of running dynamic reflective code, they are much slower. Since statically typed languages are faster and safer than dynamically typed ones, a programmer

will presumably use a dynamically typed language when the program requires runtime adaptiveness. Therefore, assessing reflective services of dynamically typed languages is an important factor to be taken into account.

We have followed a different approach to support runtime reflection over a production JIT-based virtual machine. Instead of generating extra code to simulate dynamic features over a statically typed platform, we have extended the .Net SSCLI virtual machine to directly support the reflective primitives of dynamically typed languages. The class-based model of the virtual machine has been extended with the semantics of a dynamic prototype-based model, where classes represent trait objects. New functionality has been placed in a new BCL namespace, and the semantics of some IL statements has been enhanced to support the reflective model. Depending on the language to be compiled, the compiler may use the legacy class-based model or the new prototype-based one, allowing both static and dynamic typing.

The assessment of our Reflective Rotor implementation has shown that our approach is the fastest when running reflective tests. The increase of memory consumption has been lower than the performance benefit. When running static code, we are at least 3 times faster than the rest of implementations tested, requiring at most 102% more memory resources. This benefit is because of the design of the virtual machine JIT compiler, which has been aggressively optimized to generate non-reflective code. Finally, we have also evaluated the cost of our enhancements. When running real applications that do not use reflection at all, empirical results show the performance cost is generally below 50%, using 4% more memory.

Another conclusion of our work is that the performance cost of adding reflection to the SSCLI is due to its design, strongly optimized to support statically typed languages. Its virtual machine has been optimized making assumptions of non-reflective statically-typed languages. If it had been designed to support both models, average performance of dynamic and static benchmarks would probably have been better.

Future work will be adding the rest of dynamic features of dynamically typed languages, such as dynamic inheritance and meta-classes, to give a full low-level support of these languages. These new services will be developed making use of the runtime reflective primitives described in this paper. We are also interested in including a new set of security permissions to control the reflectively manipulation of types and objects at runtime.

9 Acknowledgments

This work has been partially funded by Microsoft Research to develop the project entitled *Extending Rotor with Structural Reflection to support Reflective Languages*, awarded in 2004 in the *Second Rotor Request for Proposals*. It has also been partially funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation; project TIN2008-00276 entitled *Improving the Performance and Robustness of Dynamic Languages to develop Efficient, Scalable and Reliable Software*. The University of Oviedo has also given funds for this work, with the project UNOV-07-MB06-534.

Future work has also been funded by Microsoft Research with the project entitled *Extending dynamic features of the SSCLI*. This project is focused on adding all the dynamic services of dynamic languages to the SSCLI and defining a programming language capable of offering both static and dynamic typing.

All this work is supported by the Computational Reflection research group (`http://www.reflection.uniovi.es`) of the University of Oviedo (Spain). Binaries and source code, as well as all the benchmarks and testing code presented in this paper, can be downloaded from `http://www.reflection.uniovi.es/rrotor`.

References

[1] G. Van Rossum, F. L. Drake. The Python Language Reference Manual. Network Theory, 2003.

[2] D. Thomas, C. Fowler, A. Hunt. Programming Ruby. 2nd Edition. Addison-Wesley Professional, 2004.

[3] A. Shalit, D. Moon, O. Starbuck. The Dylan Reference Manual. Addison-Wesley, 1996.

[4] R. Ierusalimschy, L. H. de Figueiredo, W. C. Filho. Lua an extensible extension language. Software Practice & Experience 26, 6, 1996, pp. 635—652

[5] JSR 241: The Groovy Programming Language. `http://jcp.org/en/jsr/detail?id=241`

[6] O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gaelli, R. Wuyts. On the Revival of Dynamic Languages. In Proceedings of Software Composition 2005, Lecture Notes in Computer Science 3628, 2005, pp. 1-13.

[7] L. P. Deutsch, L. A. Schiffman. Efficient Implementation of the Smalltalk-80 System, In Proceedings of the 11th annual ACM Symposium on Principles of Programming Languages, 1984, pp. 297-302.

[8] U. Hölzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Technical Report: CS-TR-94-1520. Stanford University, Stanford, CA, 1994.

[9] D. Thomas, D. H. Hansson, A. Schwarz, T. Fuchs, L. Breed, M. Clark. Agile Web Development with Rails. A Pragmatic Guide. Pragmatic Bookshelf, 2005.

[10] D. Crane, E. Pascarello, D. James. Ajax in Action. Manning Publications, 2005.

[11] JSR 223. Scripting for the Java Platform. `http://www.jcp.org/en/jsr/detail?id=223`

[12] JSR 292. Supporting Dynamically Typed Languages on the Java Platform. `http://www.jcp.org/en/jsr/detail?id=292`

[13] The Pythius Website. `http://pythius.sourceforge.net`

[14] K. Böllert. On weaving aspects. In European Conference on Object-Oriented Programming (ECOOP) Workshop on Aspect Oriented Programming, 1999.

[15] Ruby Garden, Aspect Oriented Ruby. `http://www.rubygarden.org/ruby?AspectOrientedRuby`

[16] R. Hirschfeld Aspect-Oriented Programming with AspectS, In International Conference NetObjectDays, 2002, pp. 216–232.

[17] F. Ortin, J. M. Cueva. Dynamic Adaptation of Application Aspects, Journal of Systems and Software 71, 3 (May, 2004), pp. 229-243.

[18] S. Richter. Zope 3 Developer's Handbook (Sams), 2005.

[19] B. P. Pierce. Types and Programming Languages. The MIT Press. February, 2002.

[20] K. Beck. Simple Smalltalk Testing: with Patterns. 1994. `http://www.xprogramming.com/testfram.htm`

[21] E. Meijer, P. Drayton. Dynamic Typing When Needed: The End of the Cold War Between Programming Languages, In Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages, 2004.

[22] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin. Dynamic typing in a statically typed language. ACM Transactions on Programming Languages and Systems (TOPLAS) 13, 2, 1991, pp. 237-268.

[23] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin. Dynamic typing in polymorphic languages. SRC Research Report 120 (January), Digital, 1994.

[24] R. Cartwright, M. Fagan. Soft Typing, In Proceedings of Conference on Programming Language Design and Implementation, 1991, pp. 278-292.

[25] C. Chambers, D. Ungar. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language, In Proceedings of the ACM Conference on Programming Language Design and Implementation, 1989.

[26] C. Chambers. The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. PhD. Thesis. Computer Science Department, Stanford University, 1992.

[27] J. Udell. Dynamic languages and virtual machines. InfoWorld (August, 2003).

[28] P. Maes. Computational Reflection. PhD. Thesis. Laboratory for Artificial Intelligence, Vrije Universiteit, 1987.

[29] G. Kniesel, T. Rho, S. Hanenberg. Evolvable Pattern Implementations Need Generic Aspects. In RAM-SE'04 ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, 2004, pp. 111-126.

[30] D.G. Bobrow, R.G. Gabriel, J.L. White. CLOS in Context — The Shape of the Design Space, In Object-Oriented Programming — The CLOS Perspective, 1993, MIT Press.

[31] J. Kleinöder, G. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java, In Proceedings of the International Workshop on Object Orientation in Operating Systems, 1996, pp. 420-427.

[32] G. Kiczales. The Art of the Metaobject Protocol. The MIT Press, 1991.

[33] E. Tanter, J. Noy, D. Caromel, P. Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and application, 2003, pp. 27-46.

[34] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Gianini. Flickle: Dynamic Object Re-Classification. In Proceedings of the European Conference on Object-Oriented Programming ECOOP, 2001, pp. 120–149.

[35] M. Serrano. Wide Classes, In Proceedings of the European Conference on Object-Oriented Programming. Lecture Notes In Computer Science 1628, 1999, pp. 391–415.

[36] J. Siek, Walid Taha. Gradual Typing for Objects, In European Conference on Object-Oriented Programming. Lecture Notes in Computer Science 4609, 2007, pp. 2–27.

[37] S. Diehl, P. Hartel, P. Sestoft. Abstract machines for programming language implementation. Future Generation Computer Systems 16, 7, 2000, pp. 739-751.

[38] E. Meijer, J. Gough. Technical Overview of the Common Language Runtime. Technical Report. Microsoft, 2000.

[39] J. Singer. JVM versus CLR: a comparative study. In ACM Proceedings of the 2nd international conference on Principles and practice of programming in Java, 2003, pp. 167-169.

[40] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, Y. Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java. In Proceedings of the 14th European Conference on Object-Oriented Programming, 2000, pp. 362-387.

[41] The Mono Project. `http://www.mono-project.com/`

[42] DotGNU Project: GNU freedom for the Net. `http://www.dotgnu.org/`

[43] D. Stutz, T. Neward, G. Shilling. Shared Source CLI Essentials. O'Reilly, 2003.

[44] A. H. Skarra, S. B. Zdonik. Type Evolution in an Object-Oriented Database. In Research Directions in Object-Oriented Programming, 1987.

[45] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca. A Type Preserving translation of Flickle into Java. Electronic Notes in Theoretical Computer Science 62, 2002, pp. 69–82.

[46] L. Tan, T. Katayama. Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution. In Proceedings of First International Conference on Deductive and Object-Oriented Databases, 1989.

[47] J. Roddick. A Survey of Schema Versioning Issues for Database Systems. Information and Software Technology 37, 7, 1995, pp. 383-393.

[48] M. Golm, J. Kleinöder. MetaJava - A Platform for Adaptable Operating-System Mechanisms. Lecture Notes in Computer Science 1357, 1997, pp. 507-507.

[49] A. H. Borning. Classes versus Prototypes in Object-Oriented Languages, In Proceedings of the ACM/IEEE Fall Joint Computer Conference, 1986, pp. 36-40.

[50] D. Ungar, G. Chambers, B. W. Chang, U. Hölzle. Organizing Programs without Classes. Lisp and Symbolic Computation, 1991.

[51] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A. P. Black. Traits: A Mechanism for Fine-Grained Reuse. ACM Transactions on Programming Languages and Systems 28, 2 (March), 2006, pp. 331-388.

[52] M. Wolczko, O. Agesen, D. Ungar. Towards a Universal Implementation Substrate for Object-Oriented Languages. Sun Microsystems Laboratories, 1996.

[53] M. Abady, L. Cardelli.A Theory of Objects. Springer. 1998.

[54] P. Mulet, P. Cointe. Definition of a Reflective Kernel for a Prototype-Based Language- In International Symposium on Object Technologies for Advanced Software, 1993, pp. 128-144.

[55] D. Ungar, R. B. Smith. SELF: The Power of Simplicity. In OOPSLA Conference Proceedings SIGPLAN Notices 22, 12, 1987, pp. 227-242.

[56] J.B. García Perez-Schofield, E. García, F. Ortin, M. Perez. Visual Zero: A persistent and interactive object-oriented programming environment. Journal of Visual Languages and Computing 19, 3 (June), pp. 273–283, 2008.

[57] Boo Home Page. `http://boo.codehaus.org/`

[58] G. Bracha, D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In Proceedings OOPSLA '93, ACM SIGPLAN Notices 28 (October), 1993, pp. 215-230.

[59] M. Ancona, W. Cazzola. Implementing the Essence of Reflection: a Reflective Run-Time Environment. In Proceedings of the ACM Symposium on Applied Computing, pp. 1503–1507, 2004.

[60] S. Chiba. Load-Time Structural Reflection in Java. In Proceedings of European Conference on Object-Oriented Programming ECOOP, 2000, pp. 313–336.

[61] I. Welch, J.R. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting, In Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering. Lecture Notes In Computer Science 1826, 1999, pp. 155–167.

[62] R. Harper, G. Morrisett. Compiling Polymorphism using Intensional Analysis. In ACM Symposium on Principles of Programming Languages, 1995, pp. 130–141.

[63] A. Moors, F. Piessens, M. Odersky. Generics of a higher kind. ACM SIGPLAN 43, 10 (October), pp. 423–438, 2008.

[64] R. Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 3 (December), 1978, pp. 348–375.

[65] W. Landi, B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. ACM SIGPLAN Notices 39, 4 (April), 2004, pp. 473–489.

[66] A. Taivalsaari. Delegation versus concatenation or cloning is inheritance too. ACM SIGPLAN 6, 3 (July), 1994, pp. 20-49.

[67] T. Bruckschlegel. Microbenchmarking C++, C#, and Java. Dr. Dobb's Portal (June 2005).

[68] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. Communications of the ACM 27, 10, 1984, pp. 1013-1030.

[69] Benjamin Zorn CLI Benchmarks. `http://research.microsoft.com/~zorn/benchmarks/`

[70] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, R. A. Davey. A benchmark suite for high performance Java. Concurrency: Practice and Experience 12, 6, 2000, pp. 375-388.

[71] C. A. Krintz. A Collection of Phoenix-Compatible C# Benchmarks. http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks/

[72] M. Golm, J. Kleinöder. Jumping to the Meta Level: Behavioral Reflection can be fast and flexible, In Proceedings of Second International Conference on Meta-Level Architectures and Reflection, 1999, pp. 22-39.

[73] Y. Matsumoto. How Ruby sucks. In third International Ruby Conference, 2003.

[74] W. Vanderperren, D. Suvee. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In proceedings of Dynamic Aspects Workshop (DAW), 2004.

[75] M. Dahm. Byte Code Engineering. In Java-Information's Tage, 1999, pp. 267–277.

[76] The Da Vinci Machine, a multi-language renaissance for the Java Virtual Machine architecture. Sun Micryosystems OpenJDK. http://openjdk.java.net/projects/mlvm

[77] P. Kril. Sun's Da Vinci Machine broadens JVM coverage. InfoWorld. January 31, 2008.

[78] J. Hugunin. Just Glue It! Ruby and the DLR in Silverlight. In Microsoft MIX 2007, 2007. http://www.bestechvideos.com/2007/05/23/just-glue-it-ruby-and-the-dlr-in-silverlight

[79] A. Rigo. Representation-based Just-In-Time Specialization and the Psyco Prototype for Python. In Proceedings of the ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, 2004, pp. 15–26.

[80] The Psyco home page. http://psyco.sourceforge.net/

[81] The PyPy Project. http://codespeak.net/pypy/

[82] W. Cazzola. SmartReflection: Efficient Introspection in Java. Journal of Object Techonology 3, 11, 2004, pp. 117–132.