# A Hybrid Class- and Prototype-Based Object Model to Support Language-Neutral Structural Intercession

Francisco Ortin[a,*], Miguel A. Labrador[b], Jose M. Redondo[a]

[a] *University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain*
[b] *University of South Florida, Department of Computer Science and Engineering 4202 East Fowler Avenue, ENB118, Tampa, Florida, USA*

**Abstract**

**Context**: Dynamic languages have turned out to be suitable for developing specific applications where runtime adaptability is an important issue. Although .NET and Java platforms have gradually incorporated features to improve their support of dynamic languages, they do not provide intercession for every object or class. This limitation is mainly caused by the rigid class-based object model these platforms implement, in contrast to the flexible prototype-based model used by most dynamic languages.

**Objective**: Our approach is to provide intercession for any object or class by defining a hybrid class- and prototype-based object model that efficiently incorporates structural intercession into the object model implemented by the widespread .NET and Java platforms.

**Method**: In a previous work, we developed and evaluated an extension of a shared-source implementation of the .NET platform. In this work, we define the formal semantics of the proposed reflective model, and modify the existing implementation to include the hybrid model. Finally, we assess its runtime performance and memory consumption, comparing it to existing approaches.

**Results**: Our platform shows a competitive runtime performance compared to 9 widespread systems. On average, it performs 73% and 61% better than the second fastest system for short- and long-running applications, respectively. Besides, it is the JIT-compiler approach that consumes less average memory. The proposed approach of including a hybrid object-model into the virtual machine involves a 444% performance improvement (and 65% less memory consumption) compared to the existing alternative of creating an extra software layer (the DLR). When none of the new features are used, our platform requires 12% more execution time and 13% more memory than the original .NET implementation.

**Conclusion**: Our proposed hybrid class- and prototype-based object model supports structural intercession for any object or class. It can be included in existing JIT-compiler class-based platforms to support common dynamic languages, providing competitive runtime performance and low memory consumption.

*Keywords:* Structural intercession, duck typing, prototype-based object model, reflection, virtual machine, dynamic languages
*2000 MSC:* 68-04

## 1. Introduction

Dynamic languages have recently turned out to be suitable for specific scenarios such as Web development, rapid prototyping, developing systems that interact with data that change unpredictably, dynamic aspect-oriented programming, and any kind of runtime adaptable or adaptive software. The main benefit of these languages is the simplicity they offer to model the dynamicity that is sometimes required to build high context-dependent software. Common features of dynamic languages are meta-programming, reflection, mobility, and dynamic reconfiguration and distribution.

Taking Web engineering as an example, Ruby [1] has been successfully used together with the Ruby on Rails framework to create database-backed Web applications [2]. This framework has confirmed the simplicity of implementing the DRY (Don't Repeat Yourself) [3] and the Convention over Configuration [2] principles with this kind of languages. Nowadays, JavaScript [4] is being widely employed to create interactive Web applications with AJAX (Asynchronous JavaScript And XML) [5], while PHP (PHP Hypertext Preprocessor) is one of the most popular languages for developing Web-based views. Python [6] is used for many different purposes; two well-known examples are the Zope application server [7] (a framework for building content management systems, intranets and custom applications) and the Django Web application framework.

Due to the recent success of dynamic languages, statically typed languages –such as Java and .NET– are gradually incorporating more dynamic features into their platforms. Taking Java as an example, the reflection API became part of the Java platform with its release 1.1. This API offers introspection services to examine the structures of object and classes at runtime, plus object creation and method invocation –involving a substantial performance overhead. The dynamic proxy class API was added to Java 1.3. It allows defining a class at runtime that implements any interface, funneling all its method calls to an `InvocationHandler`. The Java `instrument` package (included in Java SE 1.5) provides services that allow Java agents to instrument programs running on the JVM. This package has been used to implement JAsCo, a fast dynamic AOP platform [8]. Together with other tools such as BCEL [9] and Javassist [10], these agents have also been successfully used in the implementation of application servers such as Spring Java and JBoss, obtaining good runtime performance. The Java Scripting API added to Java 1.6 permits dynamic scripting programs to be executed from, and have access to, the Java platform [11]. Finally, the Java Specification Request 292 [12] has been incorporated to the Java 1.7 Standard Edition. It adds the new `invokedynamic` opcode to the Java Virtual Machine (JVM) and the `java.lang.invoke` package to the platform [12], making it easier to implement dynamically typed languages in the Java virtual machine. Its main advantage is a user-defined linkage mechanism to postpone method call-sites resolution until runtime.

This trend has also been observed in the .NET platform. The Dynamic Language Runtime (DLR) has been included as part of the .NET framework 4.0 [13]. The DLR adds to the .NET

*Corresponding author

*Email addresses:* ortin@lsi.uniovi.es (Francisco Ortin), labrador@cse.usf.edu (Miguel A. Labrador), redondojose@uniovi.es (Jose M. Redondo)

*URL:* http://www.di.uniovi.es/~ortin (Francisco Ortin), http://www.csee.usf.edu/~labrador/ (Miguel A. Labrador)

platform a new layer that provides services to facilitate the implementation of dynamic languages over the platform [14]. Moreover, Microsoft has included the new `dynamic` type to C# 4.0, allowing the programmer to write dynamically typed code in a statically typed programming language. With this new characteristic, C# 4.0 offers direct access to code in IronPython, IronRuby and the JavaScript code in Silverlight, making use of the DLR services.

The DLR is built over the .NET virtual machine (the CLR, Common Language Runtime) to provide dynamic typing features over a statically-typed and class-based platform. It offers the common primitives provided by dynamic languages, such as structural intercession and duck typing, that the CLR does not support; it also simulates the prototype-based object model implemented by many reflective dynamic languages [15]. This extra layer over the virtual machine involves some drawbacks. The first one is that the specific features of dynamic languages are not provided for every object or class in the system. A C# programmer can only change the structure of `ExpandoObject` instances (see Section 2.3). Another limitation is that the introspective services of the platform do not take effect with `ExpandoObject`s. An additional major disadvantage is that this new software layer commonly involves a runtime performance penalty (see Section 4).

In order to overcome these limitations, we propose the addition of the specific features provided by most dynamic languages (such runtime structural intercession and duck typing) at the virtual machine level, supplying these services for any object or class. Therefore, the benefits of widely-used statically-typed platforms such as Java and .NET will be complemented with the runtime adaptiveness of dynamic languages. The use of the very same virtual machine for both types of languages will also facilitate the future interoperation between them.

The main contribution of this paper is the definition of a hybrid class- and prototype-based model valid to support structural intercession and duck typing at the virtual machine level of class-based platforms such as Java and .NET, offering these new features for any object or class. In a previous work, we extended a shared source implementation of .NET (the SSCLI, also known as Rotor) to provide some of the introspection services provided by most existing reflective languages [16]. In this work, we have extended that implementation to include the proposed hybrid class- and prototype-based model, validated with a lightweight formalization tool. Runtime performance and memory consumption of that implementation has been evaluated and compared with other existing approaches. We refer to our platform as Reflective Rotor or ЯRotor.

The rest of the paper is structured as follows. Section 2 presents the existing approaches to provide structural intercession, and our proposed model is formalized in Section 3. Section 4 presents an evaluation of runtime performance and memory consumption. Section 5 discusses the related work, and Section 6 concludes and presents future work.

## 2. Existing Approaches to Provide Structural Intercession

There are two main approaches to provide structural intercession: the class-based and the prototype-based object models. The .NET platform implements a hybrid approach, but its reflective services are not offered for every object or class (Section 2.3). As we will see, this limitation is mainly caused by the rigid class-based object model this platform actually implements in contrast to the flexible prototype-based model used by most dynamic languages. This section first defines

the basis of reflection; afterwards, the existing approaches to provide structural intercession are analyzed.

*Reflection* is the capability of a computational system to reason about and act upon itself, adjusting itself to changing conditions [17]. In a reflective language, the computational domain is enhanced with its self-representation, offering its structure and semantics as computable data at runtime. Reflection has been recognized as a suitable tool to aid the dynamic evolution of running applications, being the primary technique to obtain meta-programming, adaptiveness, and dynamic reconfiguration features of dynamic languages [18]. Computational reflection is the activity performed by a computational system when doing computation about (and by possibly affecting) its own computation [17].

*Introspection* is the reflection level that allows the inspection, but not the modification, of the program self-representation. Both Java and .NET platforms offer introspection by means of the `java.lang.reflect` package and the `System.Reflection` namespace, respectively. With these services, the programmer can obtain information about classes, objects, methods and fields at runtime. On the other hand, *intercession* is the ability of a program to modify its own execution state, including the customization of its own interpretation or meaning. Adding or removing object fields at runtime is a typical example of intercession.

Both introspection and intercession are classified as *structural* reflection when the system structure is the information reflected (offered to the programmer as data). In case the program structure is modified (i.e., structural intercession), changes will be reflected at runtime. An example of this kind of reflection is inspecting (or modifying) the structure of a class by the program itself. *Behavioral* reflection is concerned with the ability to access to the system semantics. For instance, MetaXa (formerly called MetaJava [19]) is a Java extension that offers the programmer the dynamic modification of the method dispatching mechanism.

### *2.1. Structural Intercession in Class-Based Languages*

SmallTalk and CLOS are two examples of class-based languages that provide structural intercession. Their class-based model does not provide a consistent support for every structural intercession primitive. This fact was first noticed and partially solved in the field of object-oriented database management systems [20]. In this area, objects are stored but their structure, and even their types (classes), can be altered afterwards as a result of software evolution.

*Schema evolution* is a database example that reflects the first problem of using structural intercession in the class-based model. When modifying the structure of a class, the structure of all its instances should also be updated. Class instances could be modified as soon as their class has evolved (eager) or when the object is about to be used (lazy) [21]. Dynamic evolution of class structures can produce situations such as accessing fields or methods that do not exist at a specific execution point; these situations can be detected with dynamic type checks, in order to make sure that no incorrect behavior is produced. This is how dynamic languages such as Smalltalk and CLOS provide runtime modification of classes. This mechanism has also been referred to as *type re-classification* in class-based languages, meaning that it is possible to change the class membership of an object while preserving its identity [22, 23].

The second scenario is more difficult to be modeled in class-based languages. A common intercession primitive is modifying the structure of a single object. In the class-based model, the

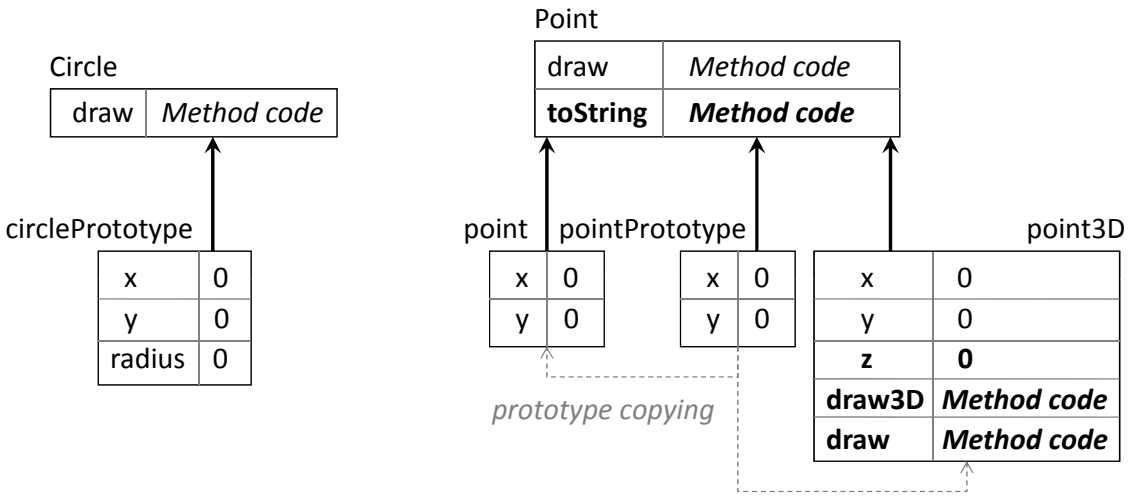


Figure 1: Structural intercession in the prototype-based object model.

modified object should be the instance of a class describing this new object structure without altering the rest of the existing instances. This problem was detected in the development of MetaXa, a reflective Java platform implementation [19]. The approach they chose was also adopted by some object-oriented database management systems: *schema versioning* [24]. A new version of the class (called "shadow" class in MetaXa) is created whenever an object structure is reflectively modified. This new class is the type of the 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 [25]. One of the conclusions of the MetaXa research project was that the class-based object-oriented model does not fit well with object-level intercession. They finally stated that *the prototype-based model would express reflective features better than class-based ones* [25].

### *2.2. Structural Intercession in Prototype-Based Languages*

The prototype-based object model suppresses the existence of classes, being the object the main abstraction of this computational model [15]. Although this model is simpler than the class-based one, there is no loss of expressiveness; i.e., any class-based program can be translated into the prototype-based model [26].

In the prototype-based model, the common behavior shared by a group of objects (i.e., methods of each class in the class-based model) is typically represented by *traits* objects [27]. These objects only collect a group of methods shared by their derived objects. The `Point` and `Circle` objects in Figure 1 can be identified as traits objects. Instead of inheritance, they provide *delegation*: when an object receives a message it cannot respond to, the object forwards the message to its traits object [28].

Common object structures can be represented in different ways. The first approach was that provided by the Self programming language, which uses *prototypes* [29]. A prototype is an object that holds a set of initialized fields representing a common structure (`pointPrototype` and

`circlePrototype` objects in Figure 1). Their translation to the class-based model is the set of instance fields declared in a class. Therefore, the creation of a new instance is performed by cloning a prototype object (Figure 1). Another approach, used by dynamic languages such as Python or Ruby, is providing a method that, using structural intercession, adds the fields of the common structure to the object passed as a parameter.

The prototype-based object-oriented computational model represents structural intercession primitives in a consistent way. Intercessive languages such as Moostrap [30], Self [29] or Lua [31] have successfully employed this model, overcoming the schema versioning problem stated in the previous section [32]. The structure (fields and methods) of a single object can be modified directly, because any object maintains its own structure and even its specialized behavior. Moreover, since shared behavior is placed in traits objects, their customization implies type adaptation (schema evolution).

Figure 1 shows an example use of intercession in the prototype-based model. Initially, there are two traits objects (`Point` and `Circle`) providing a `draw` method, and two instances representing the common structure of each type (`pointPrototype` and `circlePrototype`). The two `point` and `point3D` objects are created by copying `pointPrototype`. Using structural intercession, a new `toString` message can be provided to all the points by adding it to the base `Point` object. Then, a new `z` field and its corresponding `draw3D` method can be inserted in the single `point3D` object, without providing this functionality to the rest of points. Finally, we can override the behavior of the `draw` message for the particular `point3D` object by adding a new `draw` method, so that this specific instance will invoke `draw3D` when `draw` is called.

Many dynamic languages that support structural intercession also provide classes. However, the concept of class some of them use (e.g., Python and Ruby) is not exactly the same as the one used by class-based languages (such as Java, C# and Smalltalk). 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 without changing their classes (object-level intercession). That is, classes simply model shared behavior, the same as traits objects in the prototype-based model. Objects are responsible for storing their own structure, and they can also contain specific behavior (methods), following the prototype-based computational model.

### *2.3. Structural Intercession in the* .NET *Hybrid Model*

The .NET Framework 4.0 has included new reflective services to its platform. The new `dynamic` type added to C# 4.0 is a static type to postpone type checks until runtime. Besides, the new `ExpandoObject` class provides object-level structural intercession when a `dynamic` reference is used. The .NET approach to support structural intercession is hybrid because the platform is class-based, while `ExpandoObject` offers object-level reflection.

Figure 2 shows an example program of these dynamic features added to C# 4.0 and supported by the DLR. The first feature is duck typing: it is possible to send the `draw` message to the `figure` dynamic reference (line 22), even though `Point` and `Circle` do not have a common base interface or class except `Object`. Checking that the object provides a public `draw` method with the appropriate signature is postponed until runtime. In case no `draw` method is provided, the `RuntimeBinderException` is thrown by the DLR.

```
01: class Point {
02:   private int x, y;
03:   public void draw() {
04:     Console.WriteLine("({0}, {1})",
                          x, y);
05:     }
06: }
07:
08: class Circle {
09:   private int x, y, radius;
10:   public void draw() {
11:     Console.WriteLine("Circle in ({0},
                {1}) and radius {2}.",
                x, y, radius);
12:     }
13: }
14:
15: class Program {
16:   static void Main() {
17:     dynamic figure;
18:     if (new Random().Next(0,2) == 0)
19:       figure = new Point();
20:     else
21:       figure = new Circle();
22:     figure.draw();
23:
24:     dynamic point=new ExpandoObject();
25:     point.x = point.y = 0;
26:     point.draw = (Action)(() => {
          Console.WriteLine("({0}, {1})",
          point.x, point.y);
          });
27:     point.draw();
28:     point.ToString = (Func<string>)(() =>
          String.Format("({0}, {1})",
                        point.x, point.y));
29:     Console.WriteLine(point);
30:
31:     point.z = 0;
32:     point.move = (Action<int,int,int>)
33:          ((x,y,z) => {
                point.x = x;
                point.y = y;
                point.z = z;
           });
34:     point.move(1, 2, 3);
35:
36:     ((IDictionary<string, object>)point)
                      .Remove("draw");
37:     point.draw(); // Exception
38:
39:     showFields(figure);
40:     showFields(point);
41:     }
42:
43:   static void showFields(object obj){
44:     var fields = obj.GetType().GetFields(
              BindingFlags.NonPublic |
              BindingFlags.Instance);
45:     foreach (var field in fields)
46:       Console.WriteLine("{0}: {1}",
              field.Name,
              field.GetValue(obj));
47:     }
48: }
```

Figure 2: Example of intercession and duck typing in C# 4.0.

The second feature is object-level structural intercession: it is possible to change the structure of an object at runtime. The `point` object does not hold any field upon its creation (line 24). The `x` and `y` fields are then added by means of structural intercession (line 25). A `draw` method that uses these two fields is also added (line 26), and it is later invoked (line 27). The program makes the point three-dimensional by adding both the `z` field (line 31) and the corresponding `move` method (line 32). The `draw` method is deleted (line 36) because it does not show the new `z` coordinate. A `RuntimeBinderException` is thrown when `draw` is invoked (line 37), since it does not exist anymore.

As mentioned before, the DLR is a new software layer that provides duck typing and structural intercession services over a class-based virtual machine. Not including these services as part of the object model (i.e., at the virtual machine level) involves the following limitations:

1. Only the structure of `ExpandoObjects` can be modified. For instance, the structure of the object held by the `figure` reference in line 22 (whose type is either `Point` or `Circle`) cannot be modified with intercession because it is not an instance of `ExpandoObject`.

2. Class structures cannot evolve. In dynamic languages such as Smalltalk and CLOS, the structure of any class (e.g., `Point` or `Circle`) can be modified by means of intercession. That is not possible in the DLR.
3. Inconsistencies between duck typing and dynamic binding. In a dynamic language, the sentence in line 29 will call the new `ToString` method added to `point` in line 28. In C#, the `ToString` method in the `ExpandoObject` class is called instead. This is because dynamic binding was not implemented considering that an object may hold references to methods (as many dynamic languages do).
4. Introspection services do not work with `ExpandoObject`s. When a .NET C# programmer develops an introspective algorithm, that algorithm does not have the expected behavior when handling instances of `ExpandoObject`. In our example in Figure 2, the `showFields` method in line 43 displays the fields of any object (e.g., `figure` in line 39) using introspection. However, this behavior is not obtained when the type of the object is `ExpandoObject` (e.g., `point` in line 40): it shows the fields of the `ExpandoObject` class.
5. Deletion of members by casting objects to dictionaries. There is no mechanism to delete members of any object or class. Only members of `ExpandoObject`s can be removed, casting them to `IDictionary<string, object>` (line 36).
6. The extra layer introduced by the DLR involves a significant runtime performance penalty (a detailed evaluation is presented in Section 4).

### *2.4. Supporting Hybrid Class- and Prototype-Based Structural Intercession*

We have seen how implementing the DLR as an extra layer over the class-based virtual machine has prevented .NET from offering structural reflection for any object or class, causing the problems mentioned above. In contrast, Figure 3 shows how our proposal is aimed at including structural intercession inside the ЯRotor virtual machine implementation. The SSCLI class-based virtual machine has been extended with the prototype-based semantics to provide the intercession services of both object models. We have extended the SSCLI object model maintaining classes to take advantage of the existing optimizations, and because it implies a lower implementation effort.

Figure 3 shows how each programming language implementation must select the appropriate intercession services depending on its object model. Hybrid class- and prototype-based languages (such as C#) may select services of both models. ЯRotor provides an efficient implementation of all the intercession primitives, but it is the responsibility of each specific compiler or interpreter to select the appropriate operations. For example, the semantics of changing the type of an object is different in the two object models (detailed in Section 3.3).

The virtual machine allows calling all the intercession and duck typing services without performing static type checks. At runtime, if the requested operation cannot be provided, the corresponding exception is thrown (Section 3.3). Each programming language may implement its corresponding type system, following a *pluggable* type system approach [33]. Figure 3 illustrates different examples: an Eiffel compiler may implement its static class-based type system to provide earlier type error detection; likewise, a JavaScript interpreter can include its prototype-based dynamic type system in its own implementation on ЯRotor, or implement a powerful static dependent type system as the one described in [34].

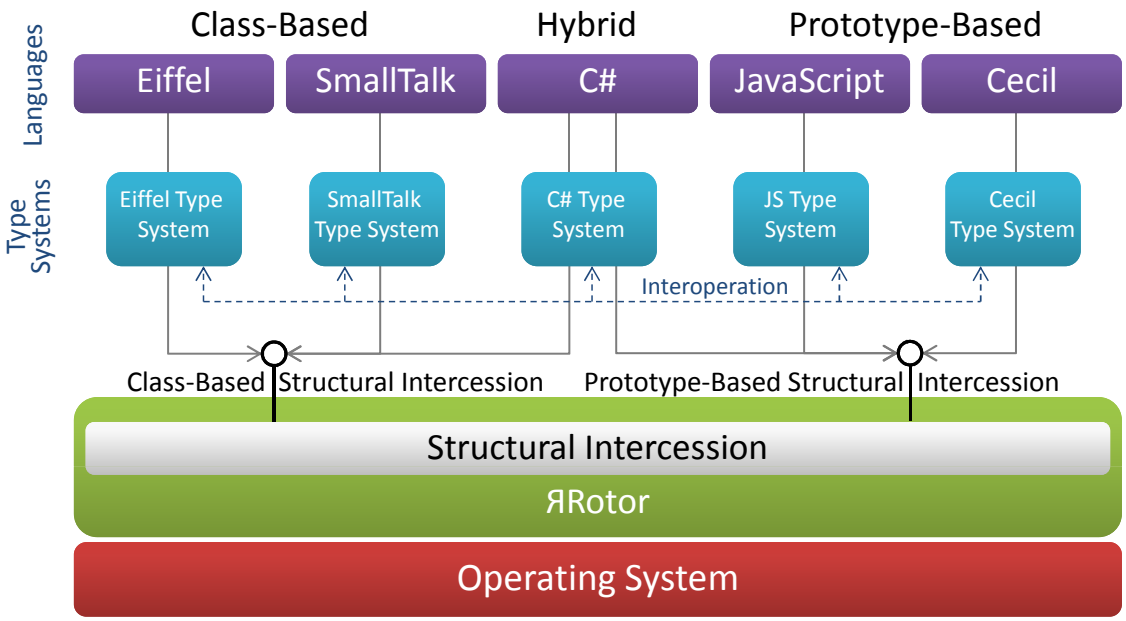


Figure 3: Supporting hybrid class- and prototype-based structural intercession.

Supporting both the class- and prototype-based models allows the execution of dissimilar programming languages over the same virtual machine. Sharing the same platform for the implementation of different languages is a valuable instrument to provide language interoperation. However, the interaction between class- and prototype-languages, and dynamically and statically typed ones, would require the use of advanced type systems. For example, *gradual* typing is an approach to allow the interoperation of dynamic and static typing, replacing type equality with type *consistency* [35]. Gradual typing also provides a robust mechanism to provide compile-time errors when hybrid statically and dynamically typed code is used. In fact, C# 4.0 includes gradual typing to support its new `dynamic` type [36]. In this article, we focus on describing the semantics of the proposed intercessive hybrid model; defining a sound type system to allow language interoperation is planned as future work (Section 6).

## 3. The ЯRotor Hybrid Class- and Prototype-Based Model

This section defines a hybrid class- and prototype-based model suitable to add structural intercession to existing class-based object-oriented platforms such as Java and .NET. For this purpose we define the syntax (Section 3.1) and semantics (Section 3.3) of the proposed object model, its properties (Section 3.4), and a summary of how we have implemented it as part of an existing class-based virtual machine (Section 3.5). The formal specification describes our hybrid class- and prototype-based intercessive model precisely, helping to discuss about it more clearly and avoiding the ambiguity of textual descriptions. The formalization describes the new features that extend the original model, avoiding the specification of these features that remained unchanged. The model has been mechanized in PLT Redex, testing its specification and properties (Section 3.4).

| | | | |
|---|---|---|---|
| Class definition expression | $e$ | ::= | $CD$ |
| Expression sequence | | \| | $e;\ \overline{e;}$ |
| Local variable expression | | \| | $x$ |
| Implicit object expression | | \| | `this` |
| Null expression | | \| | `null` |
| New object expression | | \| | `new` $C$ `(` $\overline{e}$ `)` |
| Class expression | | \| | $C$ |
| Field access expression | | \| | $e.f$ |
| Method access expression | | \| | $e.m$ |
| Method invocation expression | | \| | $e.m$ `(` $\overline{e}$ `)` \| $e$ `(` $e$ `,` $\overline{e}$ `)` |
| Variable and member assignment | | \| | ( $x$ \| $e.m$ \| $e.f$ ) = $e$ |
| Member deletion expression | | \| | `del` ( $e.m$ \| $e.f$ ) |
| Exception throwing expression | | \| | `throw` $e$ |
| Exception handling expression | | \| | `try` $e$ `catch(`$C\ x$`)` $e$ |
| Dynamic inheritance expression | | \| | $e$`.super` $((=_{\text{class}} \mid =_{\text{proto}})\ e)^{?}$ |
| Restore stack expression | | \| | $\Uparrow e\ S$ |
| Class definition | $CD$ | ::= | `class` $C$ $(:\ C)^{?}$ `{` $\overline{f}$ $\overline{M}$ `}` |
| Method definition | $M$ | ::= | $m(\overline{x})$ `{` $e$ `}` |

Figure 4: Abstract syntax of the language core.

### *3.1. Syntax*

The set of features provided by our model is an extension of the features included in Featherweight Java (FJ), the minimal Java core defined in [37]. We have added to the FJ core language the following features: class and object structural intercession (method and field addition, update and deletion), duck typing, local variables, `null` references, assignment expressions, multiple statements, the `this` reference, dynamic inheritance, exception handling, and not predefined constructors (in FJ, constructors can only initialize the object state). The cast operator was removed because we do not define a static type system. As FJ, we omit the following features of Java that do not appear to interact with structural intercession in significant ways [37]: overloading, interfaces, the `super` reference, built-in types, abstract methods and classes, `static` members and access control.

The abstract syntax of the language core is depicted in Figure 4. A program is a sequence of at least one expression, including class definitions. The meta-variables $C$ and $D$ range over class names; $m$ and $n$ range over method names and $f$ and $g$ over field names[1]; and $x$ and $y$ range over variables. As in FJ, $\overline{e}$ is shorthand for a possibly empty sequence $e_1 \ldots e_n$ (and similarly for $\overline{CD}$, $\overline{f}$, $\overline{M}$, etc.). The length of a sequence $\overline{x}$ is written $\#(\overline{x})$.

A class definition is evaluated as an expression. Each class can optionally extend another one ($^{?}$ means optionally matching the previous element), and its definition consists of a collection of fields and methods. Since our main objective is to define the dynamic semantics of a hybrid class-

[1]We separate methods and fields because classes evolve in different ways depending on the sort of member being added/modified/deleted.

```
class Point {
      x
      y
      Point(x, y) {
            this.x = x;
            this.y = y;
      }
      move(x, y) {
            this.x = x;
            this.y = y;
            this;
      }
}
class Methods {
      draw() {
            Shows (x,y) in the console
            this;
      }
      move3D(x, y, z) {
            this.x = x;
            this.y = y;
            this.z = z;
            this;
      }
}
class Point3D : Point {
      z
}
```

```
point = new Point(0, 0);
point.move(1, 2);

Point.draw = Methods.draw;
point.draw();

Point.color = null;
color = point.color;

point3D = new Point(3, 4);
point3D.z = 5;
zCoordinate = point3D.z;

point3D.move = Methods.move3D;
point3D.move(0, 0, 0);

del point3D.z;

// MissingField exception
try
  zCoordinate = point3D.z;
catch(MissingField e)
  point3D.super =class Point3D;

zCoordinate = point3D.z;
```

Figure 5: Structural intercession sample program.

and prototype-based reflective object model, we have not included the explicit type declaration of variables and fields (the same as most dynamic languages). The value returned by a method is the value of the last expression executed in its body.

Local variables do not need to be declared. As it will be described in Section 3.3, they are dynamically created (added to the topmost stack frame) by first assigning a value to them. `this` and `null` are ordinary references. An object is created using the `new` keyword, whenever a suitable constructor has been created. Classes are first-class objects; and so are object and class methods and fields: they can be passed as parameters, returned from a method, or assigned to local variables. Methods can be invoked with the typical dot operator syntax. Since they are first-class objects, they can also be called by passing to a method expression the implicit object and the method arguments as parameters. Methods and fields of both classes and objects can be added and updated with the assignment operator; they can also be removed using the `del` keyword. Exceptions are thrown with the `throw` keyword, and handled with `try`/`catch` expressions. The type of an object and the base class of another class are obtained with the `super` field. Dynamic inheritance is supported with the modification of `super` ($=_{\text{class}}$ and $=_{\text{proto}}$ represents the class- and prototype-based semantics, respectively). The non-surface $\Uparrow$ expression cannot be written by the programmer directly. It is an expression for restoring the stack, used to define the semantics of method and constructor evaluation (Section 3.3) –*S* is defined in Figure 6.

Figure 5 shows an example program in the proposed language. For the sake of brevity, we assume a predefined `Integer` class; the integer literals in the source code are instances of this class. A `Point` class is declared first. It defines two `x` and `y` fields, a constructor, and a two-dimensional `move` method. Afterwards, the `Methods` class define a `draw` method (its body is obviated) and a three-dimensional version of `move`. The `Point3D` class (derived from `Point`) is then declared with a `z` field.

The program creates a two-dimensional `point` in the origin of coordinates, and moves it to a new position. Then, a new `draw` method is added to the `Point` class, making it possible from now on to send this message to any `Point` object. By adding a new `color` field to the `Point` class, all its existing instances will have this new field (schema evolution), making `point.color` a correct expression. Another `point3D` instance is created, adding a new `z` coordinate only to this object (the other `point` instance stays unchanged). Then, a new specific three-dimensional `move` method is added to `point3D`, and it is subsequently invoked. Its `z` coordinate is deleted, throwing a `MissingField` exception when its value is requested. This exception is handled, and the type of the `point3D` object is changed to `Point3D` following the semantics of class-based languages. This type change involves adding the `z` field to `point3D`, allowing the last access to the `z` coordinate.

### *3.2. Well-Formedness*

As we have seen, explicit type declaration of variables and fields has been avoided to come closer to the representation of existing dynamic languages. Class and object members can be added and removed at runtime. Therefore, we have reduced static checking to a set of well-formedness tests, performing most type checks at runtime. Static type systems could eventually be added to provide earlier type error detection for a specific language, following the *pluggable type systems* approach [33].

Sequences of field declarations, parameter names, and method declarations are assumed to not contain duplicate names; neither do class-names. In the class declaration `class` $C : D$ `{` $\overline{f}$ $\overline{M}$ `}`, $D$ should be declared prior to $C$. At the same time, $C$ should have field names distinct from $D$ (as in FJ, we omit the Java and C# ability to allow instance variables redeclaration). On the other hand, the methods of $C$ ($\overline{M}$) may either override methods with the same names that are already present in $D$, or add new functionality to $C$.

### *3.3. Operational Semantics*

The structure of states during program evaluation is defined in Figure 6. We indicate finite mappings through $\rightharpoonup$, and undefined through *none*. We abbreviate association lists, i.e. list of pairs, in the obvious way, writing $\overline{m} \mapsto \overline{\sigma}$ for $\{m_1 \mapsto \sigma_1, \ldots, m_n \mapsto \sigma_n\}$, where $n$ is the number of pairs in the association list. When the list has only one element, we omit the curly braces (e.g., $m \mapsto \sigma$); the empty association list is represented through $\{\}$. If $H$ is a finite mapping, then $H(m \mapsto \sigma)$ is a new mapping identical to $H$ except that $m$ is overridden/added by $\sigma$. Similarly, $H \rhd m$ represents another mapping similar to $H$ except that it holds that $m \notin dom(H \rhd m)$.

A runtime state (Figure 6) is defined by an expression ($e$), a heap ($H$) and a stack ($S$)[2]. A heap maps: a) object addresses to objects, where object addresses are $\iota$; b) class names to classes,

[2]As it will be explained, the final runtime state of an uncaught exception does not contain a stack to avoid infinite recursive evaluation.

$$
\begin{array}{lcl}
RS \in \mathit{Runtime\ State} & = & \langle\, e,\, H,\, S \,\rangle \\
H \in \mathit{Heap} & = & (\, \iota \rightharpoonup \mathit{Object} \,) \cup (\, C \rightharpoonup \mathit{Class} \,) \cup (\, \sigma \rightharpoonup M \,) \\
S \in \mathit{Stack} & = & (\, \texttt{this} \cup x \,) \rightharpoonup v \\
\|C, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\| \in \mathit{Object} & = & C \ \times f \rightharpoonup \iota^n \times m \rightharpoonup \sigma^n \\
[\![C^n, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}]\!] \in \mathit{Class} & = & C^n \times f \rightharpoonup \iota^n \times m \rightharpoonup \sigma^n \\
v \in \mathit{Value} & = & \iota \cup \sigma \cup C \cup \mathit{none} \\
\multicolumn{3}{l}{\iota \in \mathit{Object\ Address},\ \ \sigma \in \mathit{Method\ Address}} \\
\multicolumn{3}{l}{\iota^n \in \iota \cup \mathit{none},\ \ \sigma^n \in \sigma \cup \mathit{none},\ \ C^n \in C \cup \mathit{none}}
\end{array}
$$

Figure 6: Structure of the runtime state.

$$
\begin{array}{lcl}
F & = & [\,] \mid \overline{v};\ F;\ \overline{e};\ \mid \texttt{new}\ C(\ \overline{v}\ F\ \overline{e}\ ) \mid F\,.\,f \mid F\,.\,m \mid F\,.\,m(\ \overline{e}\ ) \mid v\,.\,m(\ \overline{v}\ F\ \overline{e}\ ) \mid F(\ e\ ,\ \overline{e}\ ) \mid \\
 & & v(\ F\ ,\ \overline{e}\ ) \mid v(\ v\ ,\ \overline{v}\ F\ \overline{e}\ ) \mid x = F \mid F\,.\,m = e \mid v\,.\,m = F \mid F\,.\,f = e \mid v\,.\,f = F \mid \\
 & & \texttt{del}\ F\,.\,m \mid \texttt{del}\ F\,.\,f \mid \texttt{throw}\ F \mid F\,.\,\texttt{super} \mid F\,.\,\texttt{super}\ (=_{\text{class}} \mid =_{\text{proto}})\ e \mid \\
 & & v\,.\,\texttt{super}\ (=_{\text{class}} \mid =_{\text{proto}})\ F \mid \Uparrow F\ S \\
E & = & [\,] \mid \overline{v};\ E;\ \overline{e};\ \mid \texttt{new}\ C(\ \overline{v}\ E\ \overline{e}\ ) \mid E\,.\,f \mid E\,.\,m \mid E\,.\,m(\ \overline{e}\ ) \mid v\,.\,m(\ \overline{v}\ E\ \overline{e}\ ) \mid E(\ e\ ,\ \overline{e}\ ) \mid \\
 & & v(\ E\ ,\ \overline{e}\ ) \mid v(\ v\ ,\ \overline{v}\ E\ \overline{e}\ ) \mid x = E \mid E\,.\,m = e \mid v\,.\,m = E \mid E\,.\,f = e \mid v\,.\,f = E \mid \\
 & & \texttt{del}\ E\,.\,m \mid \texttt{del}\ E\,.\,f \mid \texttt{throw}\ E \mid E\,.\,\texttt{super} \mid E\,.\,\texttt{super}\ (=_{\text{class}} \mid =_{\text{proto}})\ e \mid \\
 & & v\,.\,\texttt{super}\ (=_{\text{class}} \mid =_{\text{proto}})\ E \mid \Uparrow E\ S \mid \texttt{try}\ E\ \texttt{catch}\,(C\ x)\ e
\end{array}
$$

Figure 7: Evaluation contexts.

where class names are $C$; and c) method addresses to methods, where method addresses are $\sigma$ and methods are $M$ (Figure 4). $S$ denotes the topmost stack frame, which maps local variables (including `this`) to values. Values are object or method addresses, *none* or class names. $\iota^n$, $C^n$ and $\sigma^n$ are object addresses, class names, and method addresses, respectively, that may hold the *none* value.

Objects are triples with the name of its class, a mapping from field identifiers to object references, and another one from method names to method references. Classes are similar triples, but the first optional value is the name of the base class –it is *none* when it has no super-class. We represent objects in memory through $\|C, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\|$ and classes through $[\![C^n, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}]\!]$.

The one-step reduction relation ($\longrightarrow$) is of the form $\langle e, H, S\rangle \longrightarrow \langle e', H', S'\rangle$, meaning that the expression $e$ is reduced to $e'$, where $H$ and $S$ compose the initial state and $H'$ and $S'$ the state after the reduction. We write $\xrightarrow{*}$ for the reflexive and transitive closure of $\longrightarrow$.

For conciseness, we use Felleisen-Hieb evaluation contexts [38]. Evaluation contexts ($E$ and $F$ in Figure 7) are expressions with one hole ($[\,]$) at the place of a subexpression. The expression $E[e]$ stands for the result of putting the expression $e$ into the hole of the context $E$. As shown in Figure 7, we use two kinds of evaluation contexts: $F$, for the execution of expressions without exception handling; and $E$, for expressions that may contain `try`/`catch` expressions. As it will be discussed, the separation of these two evaluation contexts facilitates the formalization of exception propagation (Figure 17).

We include class representations in the heap because their structure may evolve at runtime by means of structural intercession. This is the reason why the evaluation judgment considers the

evaluation of class definitions. When a class definition is evaluated (NE-CLASS in Figure 8), its dynamic representation is included in the heap. We also add its methods to the heap because, as we have mentioned, methods are first-class objects and some expressions are evaluated to method addresses (e.g., right hand side of the `point3D.move = Methods.move3D` assignment in Figure 5). Default values of fields are *none*.

(NE-CLASS)

$$\frac{D^n \in dom(H_1) \cup none \qquad \overline{M} = M_1, \ldots, M_n \qquad M_i = m_i(\overline{x_i})\{e_i\}^{\,i \in 1\ldots n} \qquad \sigma_i \text{ is fresh in } H_i^{\,i \in 1\ldots n}}{H_{i+1} = H_i(\sigma_i \mapsto m_i(\overline{x_i})\{e_i\})^{\,i \in 1\ldots n} \qquad H_{n+2} = H_{n+1}(C \mapsto [\![D^n, \overline{f} \mapsto none, \overline{m} \mapsto \overline{\sigma^n}]\!])}$$
$$\langle E[\texttt{class}\ \ C\texttt{:}D^n\ \{\overline{f}\ \ \overline{M}\}], H_1, S\rangle \longrightarrow \langle E[C], H_{n+2}, S\rangle$$

Figure 8: Evaluation of classes.

The `null` reference is evaluated to *none* (NE-NULL in Figure 9), modifying neither the heap nor the stack. NE-VAR describes how a variable is evaluated to its dynamically bound value in the topmost stack frame ($S$), where local variables, method parameters and `this` are stored (see NE-INV in Figure 12). NE-THIS performs the same operation when the `this` keyword is evaluated. A sequence of expressions that throw no exception is evaluated to the value returned by the last expression (NE-BLOCK).

(NE-NULL)

$$\frac{}{\langle E[\texttt{null}], H, S\rangle \longrightarrow \langle E[none], H, S\rangle}$$

(NE-VAR)

$$\frac{x \in dom(S)}{\langle E[x], H, S\rangle \longrightarrow \langle E[S(x)], H, S\rangle}$$

(NE-THIS)

$$\frac{\texttt{this} \in dom(S)}{\langle E[\texttt{this}], H, S\rangle \longrightarrow \langle E[S(\texttt{this})], H, S\rangle}$$

(NE-BLOCK)

$$\frac{}{\langle E[v_1\texttt{;}\ldots v_n\texttt{;}], H, S\rangle \longrightarrow \langle E[v_n], H, S\rangle}$$

Figure 9: Evaluation of `null`, `this`, local references and expression sequences.

When a new object is created (Figure 10), a fresh $\iota$ reference is added to the heap, associating it with a new object whose fields are taken from the $C$ class (considering inheritance). The constructor –the only method with the same name as the class– is taken from the heap, and it must have the same number of parameters as the arguments. The constructor body ($e_m$) is the following expression to be evaluated, binding the `this` reference to the new object in the topmost stack frame, and the formal parameters to the argument values.

In the reduced runtime state, a $\Uparrow \iota\ S$ expression is added after the constructor body. Its aim is restoring the stack to the state before the execution of a method ($S$), returning the value $e$ is evaluated to (NE-RSTACK). This expression facilitates the definition of imperative method invocation in small-step operational semantics [39] –substitution is not used due to the destructive assignment provided by the language.

In the proposed model, classes do not contain the list of all their inherited fields and methods. Therefore, when an object is created (Figure 10), the *fields* metafunction collects all the fields to be added to the new instance, including the inherited ones. We do not include the inherited members in class representations because it facilitates the implementation of intercession and dynamic inheritance [40].

(NE-NEW)

$$\frac{H_1(C) = [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \{\ldots, C \mapsto \sigma, \ldots\}]\!] \qquad H_1(\sigma) = C(\overline{x})\{e_m\} \\ \#(\overline{v}) = \#(\overline{x}) \qquad \iota \text{ is fresh in } H_1 \qquad H_2 = H_1(\iota \mapsto \|C, fields(H_1, C), \ \|)}{\langle E[\texttt{new}\ \ C\,(\overline{v})\,], H_1, S\rangle \longrightarrow \langle E[e_m; \Uparrow \iota\, S;], H_2, \{\texttt{this} \mapsto \iota, \overline{x} \mapsto \overline{v}\}\rangle}$$

*where*

$$\frac{}{fields(H, none) = \{\,\}} \qquad \frac{H(C) = [\![D^n, \overline{f} \mapsto \overline{\iota_f^n}, \overline{m} \mapsto \overline{\sigma^n}]\!] \\ fields(H, D^n) = \{\overline{g} \mapsto \overline{\iota_g^n}\}}{fields(H, C) = \{\overline{f} \mapsto \overline{\iota_f^n}, \overline{g} \mapsto \overline{\iota_g^n}\}} \qquad \text{(NE-RSTACK)}\ \frac{}{\langle E[\Uparrow v\, S_r], H, S\rangle \longrightarrow \langle E[v], H, S_r\rangle}$$

Figure 10: Object creation.

Figure 11 shows the evaluation of member access expressions. NE-CMACCESS and NE-OMACCESS entail the interpretation of methods as first-class objects. The NE-CMACCESS rule applies when accessing a method on a class. The *classMethod* metafunction provides a delegation-based inheritance strategy, valid for both models [41]. If a class does not implement a method, it is then searched in its base type.

(NE-CMACCESS)

$$\frac{\sigma^n = classMethod(H, C, m)}{\langle E[C.m], H, S\rangle \longrightarrow \langle E[\sigma^n], H, S\rangle}$$

(NE-OMACCESS)

$$\frac{\sigma^n = method(H, \iota, m)}{\langle E[\iota.m], H, S\rangle \longrightarrow \langle E[\sigma^n], H, S\rangle}$$

*where*

$$\frac{H(C) = [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \{\ldots, m \mapsto \sigma^n, \ldots\}]\!]}{classMethod(H, C, m) = \sigma^n} \qquad \frac{H(C) = [\![D, \overline{f} \mapsto \overline{\iota^n}, \overline{n} \mapsto \overline{\sigma^n}]\!] \qquad m \notin \overline{n} \\ classMethod(H, D, m) = \sigma^n}{classMethod(H, C, m) = \sigma^n}$$

$$\frac{H(\iota) = \|C, \overline{f} \mapsto \overline{\iota_f^n}, \{\ldots, m \mapsto \sigma^n, \ldots\}\|}{method(H, \iota, m) = \sigma^n} \qquad \frac{H(\iota) = \|C, \overline{f} \mapsto \overline{\iota_f^n}, \overline{n} \mapsto \overline{\sigma^n}\| \qquad m \notin \overline{n} \\ classMethod(H, C, m) = \sigma^n}{method(H, \iota, m) = \sigma^n}$$

(NE-CFACCESS)

$$\frac{fields(H, C) = \{\ldots, f \mapsto \iota^n, \ldots\}}{\langle E[C.f], H, S\rangle \longrightarrow \langle E[\iota^n], H, S\rangle}$$

(NE-OFACCESS)

$$\frac{H(\iota) = \|C, \{\ldots, f \mapsto \iota^n, \ldots\}, \overline{m} \mapsto \overline{\sigma^n}\|}{\langle E[\iota.f], H, S\rangle \longrightarrow \langle E[\iota^n], H, S\rangle}$$

Figure 11: Evaluation of member access.

In the proposed model, objects can hold methods. Even though methods are not placed in objects when they are created with the `new` operator (Figure 10), they can be added at runtime as in prototype-based languages such as Python, Ruby or JavaScript (Figure 13). This object representation is not appropriate for (non-hybrid) class-based languages, where methods are always placed in classes. NE-OMACCESS in Figure 11 specifies how object-level methods are accessed. The *method* metafunction starts searching the method in the object; if the object does not provide that method, it is then searched in its type, considering inheritance.

When accessing a class field, inheritance is also considered by the $fields$ metafunction in NE-CFACCESS (Figure 11). However, the access to an object field is performed by simply consulting the object (NE-OFACCESS) because, in both models, object attributes are stored in the object itself.

Figure 12 shows the two ways a method can be called. NE-INV specifies the typical object-oriented syntax. The $m$ method of the $\iota$ object is searched in the heap. Following the duck typing semantics implemented by reflective languages, the method is first searched in the object and then in the class hierarchy. This functionality is provided by the *method* metafunction defined in Figure 11. It is worth noting how this semantics overcomes the *duck typing and dynamic binding problem* mentioned in Section 2.3. If we add an `m` method to an object whose class declares a virtual `m` method, the new `m` method in the object will be called whenever this message is sent to the object. Once the appropriate method is found, the method invocation expression is reduced to the method body, mapping `this` to the object that receives the message, and the parameter variables to the actual values of the arguments. $S$ will be restored after the evaluation of $e$.

The NE-MINV rule invokes a method that has been obtained from a previous evaluation, since methods are first-class objects. The first mandatory parameter is the object on which the method is invoked. Since method access expressions (NE-CMACCESS and NE-OMACCESS in Figure 11) do not capture the object used to access the method, the object on which the method is called must be passed as the first parameter. Therefore, $e.m(\overline{e})$ is equivalent to $(e.m)(e,\overline{e})$ instead of $(e.m)(\overline{e})$.

(NE-INV)

$$\frac{\sigma = method(H,\iota,m) \qquad H(\sigma) = m(\overline{x})\{e\} \qquad \#(\overline{v}) = \#(\overline{x})}{\langle E[\iota.m(\overline{v})],H,S\rangle \longrightarrow \langle E[\Uparrow e\ S],H,\{\texttt{this} \mapsto \iota, \overline{x} \mapsto \overline{v}\}\rangle}$$

(NE-MINV)

$$\frac{H(\sigma) = m(\overline{x})\{e\} \qquad \#(\overline{v}) = \#(\overline{x})}{\langle E[\sigma(\iota,\overline{v})],H,S\rangle \longrightarrow \langle E[\Uparrow e\ S],H,\{\texttt{this} \mapsto \iota, \overline{x} \mapsto \overline{v}\}\rangle}$$

Figure 12: Evaluation of method invocation.

Figure 13 shows the inference rules for assignments of local variables, methods and fields. The NE-VASSIGN rule adds a new $x$ local variable (mapped to the $v$ value) to the topmost stack frame, or overrides its value if it had been previously placed in it.

The NE-OMASSIGN rule adds (or modifies) a method to a single object without modifying its class. This functionality can be used by both hybrid and prototype-based languages –class-based

languages do not support object-level methods, not defined in the object type. These object-level methods are considered by the method invocation rules (NE-INV and NE-MINV in Figure 12), overriding the methods defined in the object class, and providing the same mechanism for both kinds of languages. The NE-CMASSIGN rule, used in both types of languages, is similar to NE-OMASSIGN except that it adds (or modifies) a method to an existing class. We do not need to modify the derived classes because the defined semantics uses a delegation-based inheritance strategy (e.g., NE-INV in Figure 12).

Adding an object field by means of structural intercession (NE-OFASSIGN) simply modifies the structure of the object. If the object already has that field, its value is overwritten. As happens with methods (NE-OMASSIGN), this functionality is only applicable to hybrid and prototype-based languages. NE-CFASSIGN in Figure 13 shows how the behavior with classes is different. We first modify the structure of the class, mapping the $f$ field to the $\iota^n$ reference. Afterwards, all the existing instances of the modified class and its derived classes are updated the same way: if the object has no $f$ field, it is added and mapped to the $\iota^n$ reference; if the object already has an $f$ field, it maintains its value. This functionality is provided by the *addField* metafunction in Figure 13. The $C \leq_H D$ relation denotes that $C$ is a subclass of $D$ in the $H$ heap. This is how we have included the semantics of schema evolution [42] in our platform.

(NE-VASSIGN)
$$\frac{}{\langle E[x{=}v], H, S\rangle \longrightarrow \langle E[v], H, S(x \mapsto v)\rangle}$$

(NE-OMASSIGN)
$$\frac{H_1(\iota) = \|C, \overline{f} \mapsto \overline{\iota^n}, \overline{n} \mapsto \overline{\sigma_n^n}\| \qquad H_2 = H_1(\iota \mapsto \|C, \overline{f} \mapsto \overline{\iota^n}, \{\overline{n} \mapsto \overline{\sigma_n^n}\}(m \mapsto \sigma_m^n)\|)}{\langle E[\iota\,.\,m{=}\sigma_m^n], H_1, S\rangle \longrightarrow \langle E[\sigma_m^n], H_2, S\rangle}$$

(NE-CMASSIGN)
$$\frac{H_1(C) = [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \overline{n} \mapsto \overline{\sigma_n^n}]\!] \qquad H_2 = H_1(C \mapsto [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \{\overline{n} \mapsto \overline{\sigma_n^n}\}(m \mapsto \sigma_m^n)]\!])}{\langle E[C\,.\,m{=}\sigma_m^n], H_1, S\rangle \longrightarrow \langle E[\sigma_m^n], H_2, S\rangle}$$

(NE-OFASSIGN)
$$\frac{H_1(\iota_1) = \|C, \overline{g} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\| \qquad H_2 = H_1(\iota_1 \mapsto \|C, \{\overline{g} \mapsto \overline{\iota^n}\}(f \mapsto \iota_2^n), \overline{m} \mapsto \overline{\sigma^n}\|)}{\langle E[\iota_1\,.\,f{=}\iota_2^n], H_1, S\rangle \longrightarrow \langle E[\iota_2^n], H_2, S\rangle}$$

(NE-CFASSIGN)
$$\frac{H_1(C) = [\![D^n, \overline{f} \mapsto \overline{\iota_f^n}, \overline{m} \mapsto \overline{\sigma^n}]\!] \qquad H_2 = H_1(C \mapsto [\![D^n, \{\overline{f} \mapsto \overline{\iota_f^n}\}(f \mapsto \iota^n), \overline{m} \mapsto \overline{\sigma^n}]\!]) \qquad H_3 = addField(H_2, f, \iota^n, C)}{\langle E[C\,.\,f{=}\iota^n], H_1, S\rangle \longrightarrow \langle E[\iota^n], H_3, S\rangle}$$

*where*
$$\{\iota_1, \ldots, \iota_n\} = \{\iota \mid H_1(\iota) = \|D, \overline{g} \mapsto \overline{\iota_g^n}, \overline{m} \mapsto \overline{\sigma_m^n}\| \wedge D \leq_{H_1} C \wedge f \notin \overline{g}\}$$
$$\frac{H_1(\iota_i) = \|D_i, \overline{g_i} \mapsto \overline{\iota_{g_i}^n}, \overline{m_i} \mapsto \overline{\sigma_{m_i}^n}\|^{i \in 1\ldots n} \qquad H_{i+1} = H_i(\iota_i \mapsto \|D_i, \{\overline{g_i} \mapsto \overline{\iota_{g_i}^n}\}(f \mapsto \iota^n), \overline{m_i} \mapsto \overline{\sigma_{m_i}^n}\|)^{i \in 1\ldots n}}{addField(H_1, f, \iota^n, C) = H_{n+1}}$$

*and*
$$\frac{C \in dom(H)}{C \leq_H C} \qquad \frac{H(C) = [\![D, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}]\!]}{C \leq_H D} \qquad \frac{C \leq_H D \qquad D \leq_H E}{C \leq_H E}$$

Figure 13: Assignment of local variables, methods and fields.

Member deletion rules in Figure 14 perform the opposite action of member addition. Deleting a method (EN-OMDEL) or a field (EN-OFDEL) from an object (not applicable in the class-based model) is performed by simply removing the member from its representation in the heap; the same happens when a method is erased from a class in both models (EN-CMDEL). However, when a field is removed from a class (EN-CFDEL), the field is also removed from all the instances of that class and its derived classes (schema evolution). For this purpose, the $removeField(H, f, C)$ metafunction returns another heap similar to $H$ except that the $f$ field is removed from all the instances of type $C$, considering inheritance.

(NE-OMDEL)

$$\frac{H_1(\iota) = \|C, \overline{f} \mapsto \overline{\iota_f^n}, \overline{m} \mapsto \overline{\sigma_m^n}\| \quad \{\overline{m} \mapsto \overline{\sigma_m^n}\} = \{\ldots, m \mapsto \sigma^n, \ldots\} \quad H_2 = H_1(\iota \mapsto \|C, \overline{f} \mapsto \overline{\iota_f^n}, \{\overline{m} \mapsto \overline{\sigma_m^n}\} \rhd m\|)}{\langle E[\texttt{del}\ \iota\,.\,m], H_1, S\rangle \longrightarrow \langle E[\sigma^n], H_2, S\rangle}$$

(NE-CMDEL)

$$\frac{H_1(C) = [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma_m^n}]\!] \quad \{\overline{m} \mapsto \overline{\sigma_m^n}\} = \{\ldots, m \mapsto \sigma^n, \ldots\} \quad H_2 = H_1(C \mapsto [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \{\overline{m} \mapsto \overline{\sigma_m^n}\} \rhd m]\!])}{\langle E[\texttt{del}\ C\,.\,m], H_1, S\rangle \longrightarrow \langle E[\sigma^n], H_2, S\rangle}$$

(NE-OFDEL)

$$\frac{H_1(\iota_o) = \|C, \overline{f} \mapsto \overline{\iota_f^n}, \overline{m} \mapsto \overline{\sigma^n}\| \quad \{\overline{f} \mapsto \overline{\iota_f^n}\} = \{\ldots, f \mapsto \iota^n, \ldots\} \quad H_2 = H_1(\iota_o \mapsto \|C, \{\overline{f} \mapsto \overline{\iota_f^n}\} \rhd f, \overline{m} \mapsto \overline{\sigma^n}\|)}{\langle E[\texttt{del}\ \iota_o\,.\,f], H_1, S\rangle \Rightarrow \langle E[\iota^n], H_2, S\rangle}$$

(NE-CFDEL)

$$\frac{H_1(C) = [\![D^n, \overline{f} \mapsto \overline{\iota_f^n}, \overline{m} \mapsto \overline{\sigma^n}]\!] \quad \{\overline{f} \mapsto \overline{\iota_f^n}\} = \{\ldots, f \mapsto \iota^n, \ldots\} \quad H_2 = H_1(C \mapsto [\![D^n, \{\overline{f} \mapsto \overline{\iota_f^n}\} \rhd f, \overline{m} \mapsto \overline{\sigma^n}]\!]) \quad H_3 = removeField(H_2, f, C)}{\langle E[\texttt{del}\ C\,.\,f], H_1, S\rangle \longrightarrow \langle E[\iota^n], H_3, S\rangle}$$

*where*

$$\frac{\{\iota_1, \ldots, \iota_n\} = \{\iota \mid H_1(\iota) = \|D, \overline{g} \mapsto \overline{\iota_g^n}, \overline{m} \mapsto \overline{\sigma_m^n}\| \wedge D \leq_{H_1} C \wedge f \in \overline{g}\} \quad H_1(\iota_i) = \|D_i, \overline{g_i} \mapsto \overline{\iota_{g_i}^n}, \overline{m_i} \mapsto \overline{\sigma_{m_i}^n}\|^{i \in 1\ldots n} \quad H_{i+1} = H_i(\iota_i \mapsto \|D_i, \{\overline{g_i} \mapsto \overline{\iota_{g_i}^n}\} \rhd f, \overline{m_i} \mapsto \overline{\sigma_{m_i}^n}\|)^{i \in 1\ldots n}}{removeField(H_1, f, C) = H_{n+1}}$$

Figure 14: Evaluation of member deletion.

We refer to both the inheritance tree change and the type reassignment operations shown in Figure 15 as dynamic inheritance. As specified in the abstract syntax (Figure 4), each object is created by specifying a class (traits object in the prototype-based model) upon construction. The type of an object can be consulted at runtime by accessing the `super` field of the object (NE-OSUPER). In case the `super` message is sent to a class, its superclass is returned instead (NE-CSUPER).

NE-CCSUPER specifies how to change the inheritance tree in the class-based model. If the superclass of $C_1$ is changed from $D^n$ to $C_2^n$, all the fields in $D^n$ not in $C_2^n$ (including the inherited

ones) are removed from all the instances of $C_1$. Likewise, all the fields in $C_2^n$ not in $D^n$ are added to all the objects whose type is $C_1$. Finally, the inheritance relationship is changed in the heap. In the prototype-based model (NE-PCSUPER), objects are not modified because they are responsible for keeping their own fields; only the inheritance relation is updated.

Similarly, NE-COSUPER describes how to change the type of an object from $D$ to $C$ in the class-based model. The fields in $D$ not in $C$ (including inheritance) are removed from the object, and the fields in $C$ not in $D$ are added. The type of the object is finally changed in the heap. As above, the prototype-based model only requires updating the type information in heap (NE-POSUPER).

$$\text{(NE-OSUPER)}\quad \frac{H(\iota) = \|C, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\|}{\langle E[\iota\,.\,\texttt{super}], H, S\rangle \longrightarrow \langle E[C], H, S\rangle}$$

$$\text{(NE-CSUPER)}\quad \frac{H(C) = [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}]\!]}{\langle E[C\,.\,\texttt{super}], H, S\rangle \longrightarrow \langle E[D^n], H, S\rangle}$$

$$\text{(NE-CCSUPER)}\quad \frac{\begin{array}{c} C_2^n \not\leq_{H_1} C_1 \qquad H_1(C_1) = [\![D^n, \overline{f_{c_1}} \mapsto \overline{\iota^n_{c_1}}, \overline{m_{c_1}} \mapsto \overline{\sigma^n_{c_1}}]\!] \\ \{f_{r_1}, \ldots, f_{r_n}\} = dom(fields(H_1, D^n)) - dom(fields(H_1, C_2^n)) \\ H_{i+1} = removeField(H_i, f_{r_i}, C_1)^{i \in 1 \ldots n} \\ \{f_{a_1} \mapsto \iota^n_{a_1}, \ldots, f_{a_m} \mapsto \iota^n_{a_m}\} = fields(H_{n+1}, C_2^n) - fields(H_{n+1}, D^n) \\ H_{n+i+1} = addField(H_{n+i}, f_{a_i}, \iota^n_{a_i}, C_1)^{i \in 1 \ldots m} \\ H_{n+m+2} = H_{n+m+1}(C_1 \mapsto [\![C_2^n, \overline{f_{c_1}} \mapsto \overline{\iota^n_{c_1}}, \overline{m_{c_1}} \mapsto \overline{\sigma^n_{c_1}}]\!]) \end{array}}{\langle E[C_1\,.\,\texttt{super}=_{\text{class}}\, C_2^n], H_1, S\rangle \longrightarrow \langle E[C_2^n], H_{n+m+2}, S\rangle}$$

$$\text{(NE-PCSUPER)}\quad \frac{C_2^n \not\leq_{H_1} C_1 \qquad H_1(C_1) = [\![D^n, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}]\!] \qquad H_2 = H_1(C_1 \mapsto [\![C_2^n, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}]\!])}{\langle E[C_1\,.\,\texttt{super}=_{\text{proto}}\, C_2^n], H_1, S\rangle \longrightarrow \langle E[C_2^n], H_2, S\rangle}$$

$$\text{(NE-POSUPER)}\quad \frac{H_1(\iota) = \|D, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\| \qquad H_2 = H_1(\iota \mapsto \|C, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\|)}{\langle E[\iota\,.\,\texttt{super}=_{\text{proto}}\, C], H_1, S\rangle \longrightarrow \langle E[C], H_2, S\rangle}$$

$$\text{(NE-COSUPER)}\quad \frac{\begin{array}{c} H_1(\iota) = \|D, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\| \qquad \{f_{r_1}, \ldots, f_{r_n}\} = dom(fields(H_1, D)) - dom(fields(H_1, C)) \\ H_{i+1} = H_i(\iota \mapsto \|D,\ objectFields(H_i, \iota) \rhd f_{r_i}, \overline{m} \mapsto \overline{\sigma^n}\|)^{i \in 1 \ldots n} \\ \{f_{a_1} \mapsto \iota^n_{a_1}, \ldots, f_{a_m} \mapsto \iota^n_{a_m}\} = fields(H_{n+1}, C) - fields(H_{n+1}, D) \\ H_{n+i+1} = H_{n+i}(\iota \mapsto \|D, objectFields(H_{n+i}, \iota)(f_{a_i} \mapsto \iota^n_{a_i}), \overline{m} \mapsto \overline{\sigma^n}\|)^{i \in 1 \ldots m} \\ H_{n+m+2} = H_{n+m+1}(\iota \mapsto [\![C, objectFields(H_{n+m+1}, \iota), \overline{m} \mapsto \overline{\sigma^n}]\!]) \end{array}}{\langle E[\iota\,.\,\texttt{super}=_{\text{class}}\, C], H_1, S\rangle \longrightarrow \langle E[C], H_{n+m+2}, S\rangle}$$

*where*

$$\frac{H(\iota) = \|C, \overline{f} \mapsto \overline{\iota^n_f}, \overline{m} \mapsto \overline{\sigma^n}\|}{objectFields(H, \iota) = \{\overline{f} \mapsto \overline{\iota^n_f}\}}$$

Figure 15: Dynamic inheritance.

The rules above assume that during evaluation everything fits together. In contrast, many exceptional situations may arise, which we deal with by raising an exception (e.g., accessing a member of a `null` expression). That is, the expression does not evaluate to a normal value but to an exception object. For instance, the EE-VAR rule in Figure 16 throws an exception when a local variable is used and no value has been previously assigned to it. Similarly, EE-INV raises an exception when the invoked *m* method is not provided by the object that receives the message. The premises of EE-VAR and EE-INV guarantee the existence of the `VarNotDefined` and `MissingMethod` exception classes (these classes derive from `Exception` in the real implementation) in the heap, and create a new object of that type to throw it as an exception. The programmer may throw any object as an exception with the `throw` keyword and an expression evaluated to an object reference (Figure 4). For the sake of brevity, we do not include the rest of exception evaluation rules –all of them are detailed in [43].

(EE-VAR)

$$\frac{\begin{array}{c} x \notin dom(S) \\ H_2 = H_1(C_{\text{VarNotDefined}} \mapsto [\![none, , ]\!]) \\ \iota \textit{ is fresh in } H_2 \\ H_3 = H_2(\iota \mapsto \|C_{\text{VarNotDefined}}, , \|) \end{array}}{\langle E[x], H_1, S\rangle \longrightarrow \langle E[\texttt{throw}\ \iota], H_3, S\rangle}$$

(EE-INV)

$$\frac{\begin{array}{c} \nexists\, \sigma.\sigma = method(H_1, \iota, m) \\ H_2 = H_1(C_{\text{MissingMethod}} \mapsto [\![none, , ]\!]) \\ \iota_e \textit{ is fresh in } H_2 \\ H_3 = H_2(\iota_e \mapsto \|C_{\text{MissingMethod}}, , \|) \end{array}}{\langle E[\iota\,.\,m\,(\overline{v})\,], H_1, S\rangle \longrightarrow \langle E[\texttt{throw}\ \iota_e], H_3, S\rangle}$$

Figure 16: Two examples of exception evaluation inference rules.

Once an exception is thrown, it can be propagated up to the final evaluation of the program, or handled by a `try`/`catch` expression. The EP-UNCAUGHT rule in in Figure 17 shows how exceptions are propagated, causing the final evaluation of the program. $F[\texttt{throw}\ \iota]$ represents any expression without a `try`/`catch` that has `throw` $\iota$ as one of its subexpressions. The whole expression is reduced to `throw` $\iota$, representing the final evaluation of the program. Notice that the final state does not contain *S* to avoid the infinite recursive evaluation of this rule.

(EP-UNCAUGHT)

$$\frac{}{\langle F[\texttt{throw}\ \iota], H, S\rangle \longrightarrow \langle \texttt{throw}\ \iota, H\rangle}$$

Figure 17: Exception propagation.

After defining how exceptions are thrown and propagated, rules in Figure 18 show the semantics of `try`/`catch`. If the expression in the `try` block throws no exception, the `try`/`catch` statement returns the value returned by the `try` expression, and the `catch` block is not evaluated (NE-NOEXCEPT). Otherwise, if the type of the object thrown is a subtype of the type declared in the `catch` block, the exception is handled (NE-CATCH). Notice how the *E* and *F* evaluation contexts are used: the whole `try`/`catch` expression that has a `throw` subexpression as part of its

`try` block is reduced to the `catch` block ($e$), mapping $x$ to the address of the exception thrown ($\iota$). Finally, if the subtype condition is not satisfied, the exception is propagated (EP-NoCatch).

$$\text{(NE-NoExcept)}\quad \frac{}{\langle E[\texttt{try}\ v\ \texttt{catch}\ (C\ x)\ e], H, S\rangle \longrightarrow \langle E[v], H, S\rangle}$$

$$\text{(NE-Catch)}\quad \frac{H(\iota) = \|D, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\| \qquad D \leq_H C}{\langle E[\texttt{try}\ F[\texttt{throw}\ \iota]\ \texttt{catch}\ (C\ x)\ e], H, S\rangle \longrightarrow \langle E[\Uparrow e\ S], H, S(x \mapsto \iota)\rangle}$$

$$\text{(EP-NoCatch)}\quad \frac{H(\iota) = \|D, \overline{f} \mapsto \overline{\iota^n}, \overline{m} \mapsto \overline{\sigma^n}\| \qquad D \not\leq_H C}{\langle E[\texttt{try}\ F[\texttt{throw}\ \iota]\ \texttt{catch}\ (C\ x)\ e], H, S\rangle \longrightarrow \langle E[\texttt{throw}\ \iota], H, S\rangle}$$

Figure 18: Exception handling.

### *3.4. Validation with PLT Redex*

The proposed hybrid model has been validated using the PLT Redex lightweight formalization tool [44]. The process of mechanizing has helped us find errors in our semantics, particularly in the use of *none* memory addresses. Using Redex, we have tested the following two properties of our model:

> *Property 1* (One-Step Confluence). *If* $\langle e, H, S\rangle \longrightarrow \langle e_1, H_1, S_1\rangle$ *and* $\langle e, H, S\rangle \longrightarrow \langle e_2, H_2, S_2\rangle$ *then* $e_1 = e_2$, $H_1 = H_2$ *and* $S_1 = S_2$.
>
> The way fresh addresses are chosen must be deterministic depending on the heap. Besides, the runtime state $\langle e, H, S\rangle$ must be well-formed. We elide the definition of the well-formedness conditions, as the supplemental material (the Redex implementation) contains the details.
>
> *Property 2* (Progress). *If* $\langle e, H, S\rangle$ *is well-formed, then either:*
>
> - $e \in v$, *or*
> - $\langle e, H, S\rangle \longrightarrow \langle e', H', S'\rangle$, *or*
> - $\langle e, H, S\rangle \longrightarrow \langle \texttt{throw}\ \iota, H'\rangle$ *and* $\iota \in dom(H')$.
>
> The last alternative represents the evaluation of an uncaught exception, where the final state has no $S$ to avoid infinitive recursive reductions (see EP-Uncaught in Figure 17).

Each property has been tested with two rounds of 15,000 random tests: the first one with initial empty stack and heap, and the second one with random heap and stack. For the second kind of scenarios, the initial random runtime state is checked for well-formedness and converted into a well-formed one to increase the test coverage. Besides, we have manually coded 585 additional

tests. The average coverage of normal evaluation (NE-*), exception propagation (EP-*) and exception evaluation (EE-*) rules were 986, 7,031 and 526, respectively, with a minimum coverage of 2.

### *3.5. Implementation*

We have implemented the semantics of the proposed hybrid model as part of a widespread class-based virtual machine. We have chosen the virtual machine of the Microsoft .NET platform because of its design focused on supporting a wide number of languages [45, 46], and its support of dynamic languages by means of the DLR [13]. Extending the .NET platform to provide a direct support for dynamically typed languages facilitates future interoperation with existing languages and any .NET application and component. Since one of our objectives is to show the efficiency of the proposed model, we looked for a production JIT-based virtual machine implementation. We took the SSCLI (Shared Source Common Language Infrastructure) implementation of the Microsoft .NET platform because, although there exist other implementations (such as Mono [47] or DotGNU Portable.NET [48]), the SSCLI is closer to the commercial virtual machine implementation: the CLR. Therefore, it could give us an estimate of the runtime performance obtained if the semantics is included in the commercial version (see Section 4.4), and compare it with the DLR –the existing approach to support dynamic languages on .NET.

We have modified our previous extension of the SSCLI that supported several structural introspection services [16], including the hybrid object model specified in Section 3. These introspective services have been added to the Base Class Library (BCL) in a new namespace called `System.Reflection.Structural`. All of them were modified to include the proposed model. These are the most significant reflective primitives added to that namespace (all of them are `static` methods of the `NativeStructural` utility class):

- `addField` and `addMethod`: Provide the addition of fields and methods to an object (NE-OMASSIGN and NE-OFASSIGN) or class (NE-CMASSIGN and NE-CFASSIGN).
- `removeField` and `removeMethod`: Delete fields and methods from objects (NE-OFDEL and NE-OMDEL) and classes (NE-CFDEL and NE-CMDEL).
- `getField` and `getMethod`: Return a field and method of an object (NE-OFACCESS and NE-OMACCESS) or class (NE-CFACCESS and NE-CMACCESS), considering the hybrid object model defined.
- `invoke`: Performs the dynamic invocation of methods as described in NE-INV and NE-MINV. The new `RuntimeStructuralMethodInfo` class (derived from `MethodInfo`) has also been added to allow the representation of methods as first-class objects.
- `setSuper`: Overloaded method that implements the type change primitive (NE-COSUPER and NE-POSUPER) and the modification of the inheritance tree (NE-PCSUPER and NE-CCSUPER) for both object models.

Additionally, we have extended the semantics of some virtual machine opcodes, changing the binary code that the JIT compiler generates at runtime:

– `ldfld` and `ldflda` load object fields onto the stack (NE-CFAccess), and `ldsfld` and `ldsflda` push class fields (NE-OFAccess).

– Method pointers of classes (NE-CMAccess) and objects (NE-OMAccess) are loaded onto the stack with the `ldftn` and `ldvirtftn` opcodes.

– `call` and `callvirt` implement method invocation, following the semantics described in NE-Inv.

– `stfld` assigns fields to objects (NE-OFAssign) and `stsfld` to classes (NE-CFAssign).

– The type of an object is dynamically obtained with `refanytype` (NE-OSuper).

– Extending the original object model has also demanded the modification of other opcodes such as `castclass` (casts an instance to type), `isinst` (checks the type of an object) and `ldtoken` (loads the reflective representation of a method, field or type).

The existing `Reflection` namespace has been modified to take into account the hybrid object model proposed in this paper. New optimizations such as caching member pointers to reduce the cost of delegation inheritance, reducing the use of strings, and inlining those functions frequently called by the JIT-compiled code, have also been added to [16]. Finally, the implementation has been highly refactored to facilitate the inclusion of new functionalities. The detailed information about the implementation can be consulted in [43, 16].

## 4. Evaluation

This section is aimed at evaluating whether the proposed model can be included in a JIT-compiler virtual machine, obtaining competitive runtime performance and memory consumption for both short- and long-running applications. The first subsection outlines the experimental methodology used. For each kind of benchmark, we present and discuss data of the runtime performance and memory consumption (the whole evaluation data can be consulted in [43]).

### *4.1. Methodology*

The methodology comprises a description of the language implementations and the benchmark suites used in the evaluation, together with a description of how data is measured and analyzed.

#### *4.1.1. Selected Language Implementations*

We have considered some programming languages and platforms to be compared with our implementation. Since our proposed semantics focuses on object-oriented reflective languages, we have selected different implementations of the Python, Ruby and JavaScript dynamic languages because of their wide popularity and utilization at present. Distinct implementations of each language have been selected, including existing JIT-compiler versions. We have also used the last version of C# (4.0) that exploits the DLR services. These are the specific implementations:

– CPython 2.7.3 and 3.2.3 for Windows (commonly referred as simply Python). These are the most widely used Python implementations; they are called CPython because they have been developed in C. There are two parallel versions because of compatibility issues.

– Jython 2.5.2 over the Java HotSpot VM 1.7 update 3 64 bits for Windows. A 100% pure Java implementation of the Python programming language. It is seamlessly integrated with the Java platform.

– IronPython 2.7.2.1 over the .NET Framework 4.0 for 32 bits. It is an open-source implementation of the Python programming language which is tightly integrated with the .NET Framework, targeting the DLR. It compiles Python programs into IL (Intermediate Language) bytecodes that run on both Microsoft's .NET and the Mono open source platform.

– PyPy 1.9 for Windows (32 bits), an alternative implementation of Python that provides JIT compilation, memory usage optimizations, and full compatibility with CPython 2.7.2. PyPy implements a tracing JIT compiler to optimize program execution at runtime, generating dynamically optimized machine code for the hot code paths of commonly executed loops [49]. Since PyPy is a Python interpreter written in Python, tracing JIT compilation is applied to the Python interpreter itself.

– Ruby 1.9.3-p194 for Windows. Ruby is a dynamic open-source programming language with a focus on simplicity and productivity. Its 1.9 version has changed the C interpreter implementation to a compiler that generates code for the YARV (Yet another Ruby VM) virtual machine, involving an important performance benefit. YARV does not implement JIT compilation.

– IronRuby 1.1.3 over the .NET Framework 4.0 for 32 bits. It is an open-source implementation of Ruby 1.9 that obtains the benefits of the DLR platform.

– V8 JavaScript Engine 3.15. V8 is the Google's open source JavaScript engine used in Google Chrome, which can run standalone or embedded into any C++ application. V8 implements a runtime adaptive JIT compiler that dynamically reoptimizes the generated code based on heuristics of the code execution profile.

– C# 4.0/DLR. The last version of C#, making use of its new features provided by the DLR: `dynamic` references and `ExpandoObject`s. This language allows writing both dynamically and statically typed one.

We also considered Objective-C because it provides both static and dynamic typing, introspection, and it is commonly natively compiled. However, we finally did not include it in our evaluation because of its limited support of intercession services (included in its runtime library): instance variables cannot be added to existing classes, only to new ones (`class_addIVar`); new methods can be added to neither classes nor objects (`class_addMethod`); members cannot be erased; object structure modification is not provided; modifying a superclass does not change the structure of classes and objects (`class_setSuperclass`); and changing the object type removes the values of instance variables (`object_setClass`).

These implementations have been evaluated together with ЯRotor compiled in the *free* operation mode, without debug information and with the highest degree of code optimization. The source code has been written in C#, using the new intercession services provided by the `Structural` namespace (Section 3.5). When a suitable IL instruction can be used instead, we replace the BCL invocation with the corresponding IL opcode to obtain better runtime performance –a compiler that performs this operation automatically has not been implemented yet.

*4.1.2. Selected Benchmarks*

We have used different benchmark suites to evaluate the efficiency of our implementation. The benchmarks have been divided into three different sets, regarding the features we want to measure:

- Structural Intercession. We have developed a set of micro-benchmarks to measure the efficiency of the structural intercession primitives provided by most dynamically typed languages. We have also measured the execution of two existing Python benchmarks (some tests of the Pybench benchmark and the Parrot benchmark) that make extensive use of structural intercession.

- Dynamic Typing. These tests are aimed at evaluating the performance of the dynamic typing features we have incorporated into the .NET virtual machine. We have measured four different benchmarks where all the variables are dynamically typed, but no structural intercession is used at all. Two of them (Pystone and Pybench) are specific for dynamic languages. The third one (Shootout) is used to measure performance of both statically and dynamically typed languages. Lastly, we have taken a fourth benchmark designed for statically typed languages (Bruckschlegel) and modified it to become dynamically typed (all the references are declared as dynamic). By using this four different types of benchmarks, we have tried to measure a wide set of dynamic typing scenarios.

- Statically Typed Code. We have compared our implementation with the original SSCLI, the CLR and the DLR. The objective is twofold: first, to evaluate the cost of our extension compared with the original SSCLI; second, to estimate the current performance penalty in the .NET Framework when using dynamically typed references vs. statically typed ones, and contrast it with our approach. For this evaluation, we have taken four existing benchmarks (Java Grande, Ben Zorn, Shootout and Bruckschlegel) where neither reflection nor duck typing is used.

The purpose of using different benchmarks and languages is to compare the performance of the same operations in different implementations, not to estimate which language performs better. For this purpose, we have taken either Python (Pystone, Parrot and Pybench) or C# (Java Grande, Ben Zorn, Shootout and Bruckschlegel) programs, and manually translated them into the rest of languages. Although this translation might introduce a bias on the runtime performance of the translated programs, we have thoroughly checked that the same operations are executed in all the implementations. In those cases that the source program uses a specific language feature, we have replaced its source code to ensure that every version performs the same operations. Only the Python Parrot benchmark has been slightly modified to avoid the use of lambda functions and the `yield` statement.

*4.1.3. Data Analysis*

We have followed the methodology proposed in [50] to evaluate the runtime performance of applications, including those executed on virtual machines that provide JIT compilation. In this methodology, two approaches are considered: 1) *start-up* performance is how quickly a system can run a relatively short-running application; 2) *steady-state* performance concerns long-running applications, where start-up JIT compilation does not involve a significant variability in the total running time, and hot-spot dynamic optimizations have been applied.

To measure start-up performance, a two-step methodology is used:

1. We measure the execution time of running multiple times the same program. This results in $p$ (we have taken $p = 30$) measurements $x_i$ with $1 \leq i \leq p$.
2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is calculated using the *Student's t*-distribution because we took $p = 30$ [51]. Therefore, we compute the confidence interval $[c_1, c_2]$ as:

   $$c_1 = \overline{x} - t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}} \qquad c_2 = \overline{x} + t_{1-\alpha/2;p-1}\frac{s}{\sqrt{p}}$$

   Being $\overline{x}$ the arithmetic mean of the $x_i$ measurements, $\alpha = 0.05(95\%)$, $s$ the standard deviation of the $x_i$ measurements, and $t_{1-\alpha/2;p-1}$ defined such that a random variable $T$, that follows the *Student's t*-distribution with $p-1$ degrees of freedom, obeys $Pr[T \leq t_{1-\alpha/2;p-1}] = 1-\alpha/2$. The data provided is the mean of the confidence interval plus the confidence internal.

In the start-up methodology, the $x_i$ measurements represent the execution time of the whole process. Therefore, this evaluation method includes the start-up execution time of the language processor itself, helping to analyze the appropriateness of each processing technique (e.g., JIT-compilation) for executing short-running applications. In contrast, the steady-state methodology only considers the execution time of the benchmark, not the whole process. The steady-state methodology comprises the following four steps:

1. Each application (program) is executed $p$ times ($p = 30$), and each execution performs at least $k$ ($k = 10$) different iterations of benchmark invocations, measuring each invocation separately. We refer $x_{ij}$ as the measurement of the $j^{th}$ benchmark iteration of the $i^{th}$ application execution.
2. For each $i$ invocation of the benchmark, we determine the $s_i$ iteration where steady-state performance is reached. The execution reaches this state when the coefficient of variation ($CoV$, defined as the standard deviation divided by the mean) of the last $k$ iterations (from $s_{i-k+1}$ to $s_i$) falls below a threshold (2%).
   To avoid an influence of the previous benchmark execution, a full heap garbage collection is done before performing every benchmark invocation. Garbage collection may still occur at benchmark execution, and it is included in the measurement. However, this method reduces the non-determinism across multiple invocations due to garbage collection kicking in at different times across different executions.

3. For each application execution, we compute the $\overline{x_i}$ mean of the $k$ benchmark iterations under steady state:

$$\overline{x_i} = \frac{\sum_{j=s_{i-k+1}}^{s_i} x_{ij}}{k}$$

4. Finally, we compute the confidence interval for a given confidence level (95%) across the computed means from the different application invocations using the *Student's* $t$-statistic described above. The overall mean is computed as $\overline{x} = \sum_{i=1}^{p} \overline{x_i}/p$. The confidence interval is computer over the $\overline{x_i}$ measurements.

*4.1.4. Data Measurement*

To measure execution time of each benchmark invocation (in the steady-state methodology), we have instrumented the applications with code that registers the value of high-precision time counters provided by the Windows operating system. This instrumentation calls the native function `QueryPerformanceCounter` of the `kernel32.dll` library. This function returns the execution time measured by the operating system Performance and Reliability Monitor [52]. We measure the difference between the beginning and the end of each benchmark invocation to obtain the execution time of each benchmark run.

The memory consumption has been measured following the start-up methodology, determining the memory used by the whole process. For that purpose, we have used the maximum size of working set memory employed by the process since it was 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 application to be used 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. The `PeakWorkingSet` has been measured with explicit calls to the services of the Windows Management Instrumentation infrastructure [53].

All the tests were carried out on a 2.67 GHz Intel I7 920 system with 8 GB of RAM running an updated 64-bit version of Windows 7 Professional SP1. The benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded (until the CPU usage falls below 2% and remains at this level for 30 seconds). If the $L_1$ and $L_2$ languages run the same benchmark in $T$ and $1.5\times T$ milliseconds, respectively, we say that $L_1$ is 50% (or 1.5 times) faster than $L_2$, runtime performance of $L_1$ is 50% (or 1.5 times) higher than $L_2$, $L_2$ requires 50% (or 1.5 times) more execution time that $L_1$, or the performance benefit of $L_1$ compared to $L_2$ is 50% –the same for memory consumption. To compute average percentages, factors and orders of magnitude, we use the geometric mean.

*4.2. Structural Intercession*

To evaluate the efficiency of intercession, we have created a synthetic micro-benchmark that executes the typical intercession primitives. We invoke each primitive in a loop of 10,000 iterations. The objective of this micro-benchmark is to evaluate the efficiency of the primitives we have added to the .NET platform in isolation, so that we can compare our implementation with

existing approaches. Afterwards, we measure two existing applications that combine the usage of intercession with non-reflective code to obtain a specific purpose.

Table 1 and 2 show the execution times of our micro-benchmark in milliseconds, using the start-up and steady-state methodologies respectively. This micro-benchmark consists of 22 reflection tests: tests 1-10, 15-16, and 19-20 are relative to intercession, tests 11-14 and 17-18 measure duck typing, and tests 21 and 22 evaluate changing the inheritance tree and the type of an object, respectively (see [43] for a description of each test). As shown in both tables, the DLR does not provide intercession and duck typing when dealing with classes: only single instances (of `ExpandoObjects`) can be modified; likewise, `dynamic` can be used to send messages only to objects (not to classes). The DLR and Ruby do not support dynamic inheritance (tests 21 and 22).

| Test | ЯRotor | CPy2 | CPy3 | Jython | IronPython | PyPy | Ruby | IronRuby | V8 | DLR |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 129.44 | 161.77 | 166.67 | 14,283.53 | 6,284.39 | 1,261.77 | 223.55 | 1,462.22 | 178.77 | 1,396.73 |
| 2 | 111.47 | 168.12 | 173.73 | 13,547.63 | 6,590.91 | 1,366.83 | 427.66 | 3,327.52 | 146.17 | 1,222.40 |
| 3 | 54.54 | 152.00 | 162.70 | 15,127.50 | 5,899.14 | 496.53 | 247.51 | 2,262.20 | 129.60 | |
| 4 | 50.44 | 169.25 | 173.00 | 13,463.33 | 5,779.26 | 458.77 | 265.55 | 2,577.68 | 168.67 | |
| 5 | 65.80 | 278.00 | 302.00 | 30,332.17 | 19,072.09 | 744.63 | 318.84 | 2,923.54 | 290.77 | |
| 6 | 64.00 | 300.00 | 287.03 | 30,695.63 | 18,660.07 | 703.50 | 350.59 | 3,384.19 | 399.03 | |
| 7 | 59.40 | 145.92 | 139.58 | 12,370.33 | 6,469.64 | 14,925.67 | 61.00 | 727.87 | 132.20 | 560.22 |
| 8 | 59.37 | 145.17 | 147.07 | 15,523.67 | 6,604.84 | 14,750.33 | 87.84 | 863.85 | 109.00 | 556.40 |
| 9 | 65.17 | 141.38 | 141.33 | 16,227.70 | 6,378.83 | 334.00 | 114.41 | 878.95 | 167.33 | |
| 10 | 61.67 | 164.20 | 137.75 | 15,846.57 | 6,816.46 | 336.57 | 115.61 | 873.45 | 137.90 | |
| 11 | 0.33 | 138.77 | 116.70 | 12,881.53 | 12,925.08 | 310.43 | 1,102.06 | 2,876.66 | 110.00 | 4,776.00 |
| 12 | 249.35 | 193.07 | 140.25 | 12,636.07 | 1,605.43 | 313.50 | 725.04 | 3,756.38 | 125.00 | 1,295.50 |
| 13 | 39.37 | 183.63 | 152.93 | 15,716.77 | 1,005.76 | 905.83 | 117.61 | 1,032.46 | 125.00 | 509.75 |
| 14 | 44.14 | 126.80 | 119.82 | 17,717.83 | 12,129.69 | 342.33 | 74.00 | 905.84 | 134.37 | |
| 15 | 39.74 | 50.22 | 45.00 | 4,006.30 | 281.75 | 203.75 | 53.30 | 370.02 | 109.00 | 67.76 |
| 16 | 38.87 | 64.20 | 59.60 | 3,624.70 | 1,639.09 | 148.75 | 53.60 | 387.62 | 109.00 | |
| 17 | 27.13 | 50.00 | 46.83 | 3,318.13 | 273.88 | 156.00 | 30.64 | 688.71 | 31.67 | 77.13 |
| 18 | 19.10 | 30.00 | 33.92 | 1,849.40 | 4,801.07 | 115.20 | 144.91 | 744.88 | 63.00 | 77.17 |
| 19 | 8.00 | 39.00 | 40.57 | 3,147.00 | 257.05 | 15,706.87 | 32.00 | 338.35 | 26.37 | 33.00 |
| 20 | 6.00 | 32.37 | 38.73 | 2,928.37 | 1,533.68 | 70.47 | 22.00 | 238.68 | 15.00 | |
| 21 | 47.93 | 153.90 | 6,845.17 | 169.50 | 87.84 | 10.00 | | | 1,350.80 | |
| 22 | 13.00 | 85.80 | 63.97 | 299.10 | 62.40 | 8.40 | | | 124.33 | |
| **Total** | **1,254** | **2,973** | **9,534** | **255,712** | **125,157** | **53,670** | **4,567** | **30,621** | **4,182** | **10,572** |

Table 1: Start-up execution time (ms) of the micro-benchmark.

Table 1 shows how ЯRotor is, on average, the fastest implementation in the start-up methodology: 3.24, 3.69, 200, 70, 12.3, 3.7, 30.9, 3.51 and 12.1 times faster than CPython 2, CPython 3, Jython, IronPython, PyPy, Ruby, IronRuby, V8 and the DLR, respectively (Figure 19). In all the primitives, except for the 12$^{th}$ one (accessing class fields) and the two dynamic inheritance operations (21$^{st}$ and 22$^{nd}$ primitives), ЯRotor obtains the lowest start-up execution time. When accessing `static` members (12$^{th}$ primitive), runtime performance of the SSCLI is significantly slower than its counterparts [16]. PyPy is the only implementation that performs better than ЯRotor for dynamic inheritance primitives, due to the runtime program optimizations implemented by its tracing JIT compiler [49].

In the steady-state methodology (Table 2), ЯRotor also obtains the best average runtime performance: 6.09, 6.87, 295, 31.1, 17.4, 9.21, 60.3, 1.42 and 15.52 times faster than CPython 2, CPyhon 3, Jython, IronPython, PyPy, Ruby, IronRuby, V8 and the DLR, respectively. However, V8 performs better than ЯRotor in 10 tests. This improvement is due to the suitability of its adaptive JIT-compiler for long-running applications. The JIT-compiler of the SSCLI does not per-

form dynamic optimizations of the code generated, and thus the improvement for long-running applications is not as high as the one obtained by V8.

| Test | ЯRotor | CPy2 | CPy3 | Jython | IronPython | PyPy | Ruby | IronRuby | V8 | DLR |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 82.22 | 155.70 | 143.90 | 14,044.78 | 796.40 | 1,073.67 | 246.38 | 1,497.35 | 42.40 | 1,239.02 |
| 2 | 72.29 | 171.46 | 151.05 | 13,412.10 | 830.85 | 1,086.88 | 387.16 | 3,397.18 | 54.40 | 1,186.82 |
| 3 | 42.23 | 160.77 | 141.70 | 13,951.60 | 789.26 | 369.50 | 349.40 | 2,346.16 | 15.00 | |
| 4 | 37.10 | 165.27 | 153.35 | 13,314.20 | 855.46 | 380.63 | 340.52 | 2,878.73 | 22.00 | |
| 5 | 50.50 | 278.45 | 267.00 | 26,068.23 | 1,537.24 | 646.05 | 424.99 | 3,356.04 | 281.20 | |
| 6 | 45.72 | 295.05 | 281.88 | 26,097.20 | 1,603.79 | 664.40 | 502.33 | 3,914.99 | 50.20 | |
| 7 | 40.35 | 140.97 | 134.00 | 12,371.32 | 872.23 | 8,103.63 | 59.45 | 793.51 | 65.40 | 573.05 |
| 8 | 43.07 | 143.60 | 137.18 | 11,724.13 | 847.60 | 8,444.13 | 84.78 | 905.85 | 53.00 | 559.05 |
| 9 | 37.50 | 140.65 | 136.72 | 12,629.36 | 841.65 | 334.40 | 133.64 | 968.08 | 26.60 | |
| 10 | 41.17 | 143.63 | 137.15 | 11,622.30 | 844.25 | 334.85 | 137.61 | 994.61 | 30.80 | |
| 11 | 0.33 | 121.35 | 114.27 | 12,239.63 | 1,359.51 | 294.21 | 1,229.74 | 3,464.57 | 9.00 | 522.91 |
| 12 | 235.60 | 147.18 | 137.16 | 13,267.55 | 1,339.34 | 309.70 | 843.95 | 231.31 | 79.80 | 4.50 |
| 13 | 30.56 | 160.47 | 147.35 | 13,352.30 | 812.23 | 812.50 | 156.93 | 1,171.39 | 59.80 | 516.51 |
| 14 | 27.20 | 122.95 | 116.97 | 11,913.67 | 750.63 | 296.54 | 84.34 | 1,056.48 | 14.20 | |
| 15 | 20.21 | 46.38 | 43.60 | 3,288.52 | 212.09 | 172.36 | 64.51 | 428.37 | 7.20 | 64.67 |
| 16 | 19.26 | 59.93 | 55.43 | 3,224.88 | 226.55 | 129.92 | 57.98 | 445.04 | 3.40 | |
| 17 | 0.70 | 48.25 | 44.30 | 3,587.85 | 218.58 | 149.50 | 35.47 | 771.02 | 3.00 | 34.88 |
| 18 | 0.10 | 30.79 | 33.39 | 1,515.80 | 1,271.86 | 98.38 | 140.89 | 260.19 | 3.00 | 64.67 |
| 19 | 5.82 | 38.88 | 36.87 | 3,202.12 | 234.24 | 6,838.95 | 38.50 | 405.69 | 5.00 | 30.10 |
| 20 | 5.18 | 30.60 | 30.27 | 3,219.10 | 209.67 | 69.17 | 26.61 | 255.47 | 6.00 | |
| 21 | 57.67 | 129.06 | 6,620.60 | 20.37 | 63.96 | 5.86 | | | 1,395.00 | |
| 22 | 12.00 | 72.68 | 62.40 | 30.64 | 53.82 | 0.40 | | | 26.20 | |
| **Total** | **907** | **2,804** | **9,127** | **224,098** | **16,571** | **30,616** | **5,345** | **29,542** | **2,253** | **4,796** |

Table 2: Steady-state execution time (ms) of the micro-benchmark.

Together with our micro-benchmark that evaluates the efficiency of reflective primitives, we have also evaluated two existing programs that make use of intercession. The first one is the Parrot benchmark [54], a complex program 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 traversal algorithms at runtime by means of structural intercession. The second program we have selected is Pybench [55], a Python benchmark designed to measure the performance of standard Python implementations. Pybench is composed of a collection of tests that measures different aspects of the Python programming language. In this section, we have measured the 6 tests that make use of intercession (the rest are evaluated in Section 4.3).

Figure 19 shows the average execution time, relative to ЯRotor, of these two benchmarks (together with the average execution time of the micro-benchmark) –when a value is much higher than the rest, its representation has been cut to improve the visualization of the figure, and values are displayed over each bar. ЯRotor provides the best runtime performance in all the benchmarks with the exception of PyBench, for which V8 requires 66% the execution time used by ЯRotor (both start-up and steady-state). V8 uses an advanced adaptive JIT compiler, which optimizes code at runtime based on profiling information.

Figure 20 displays the average memory usage relative to ЯRotor for the benchmarks analyzed in this section –together with those analyzed in the following one. In the three benchmarks that use structural intercession, memory consumption of the interpreted implementations is lower than the JIT-compilation approaches. On average, CPython 2, CPython 3 and Ruby consume 58.86%, 75% and 73.43% the memory used by our platform. V8 requires 85.46% the memory of ЯRotor;

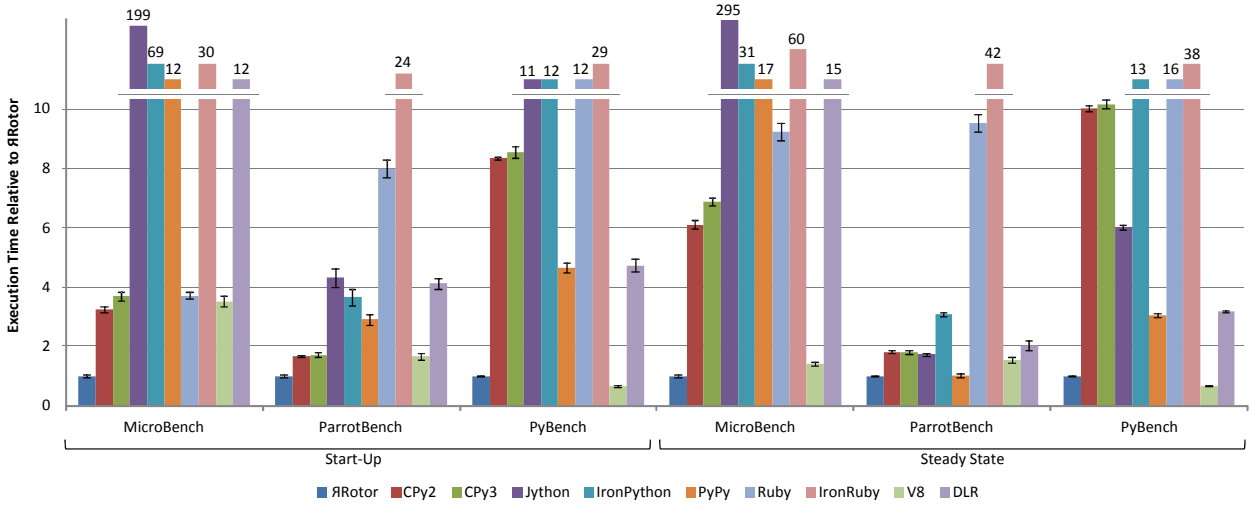


Figure 19: Execution time of the intercession benchmarks relative to ЯRotor.

the second JIT-compiler platform (after V8) with the lowest memory requirements. Jython, IronPython, PyPy, IronRuby and the DLR consumes 1,431%, 244%, 222%, 237% and 11.68% more memory than ЯRotor.

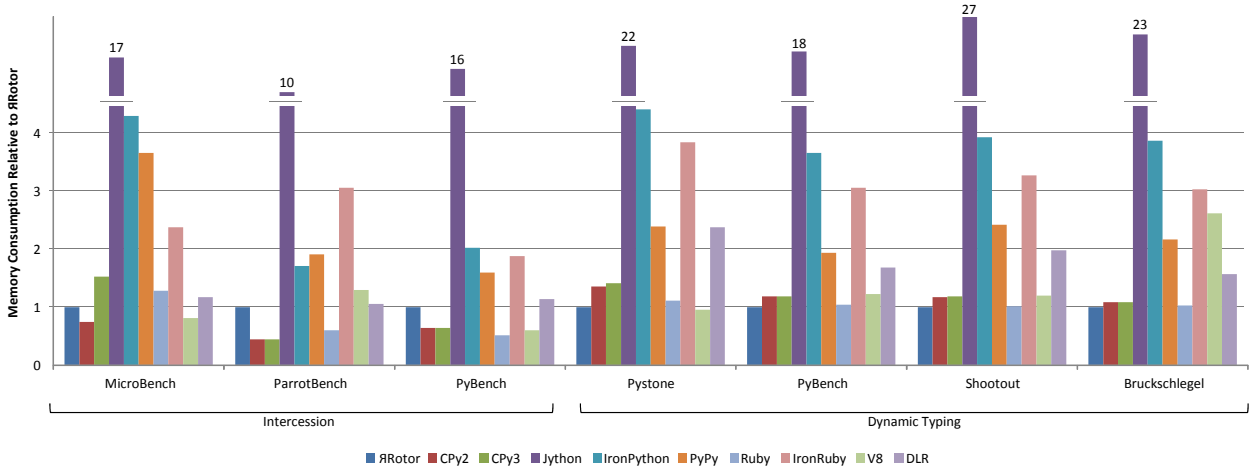


Figure 20: Memory consumption of the intercession and dynamic typing benchmarks relative to ЯRotor (confidence intervals are not shown because all of them are below 2%).

In this first evaluation section, we have seen how our implementation obtains the best average performance in the micro-benchmark that synthetically measures the common intercession primitives. In this scenario, the interpreted-based approaches and V8 perform better that the rest of JIT-compiler implementations. However, when non-intercessive code is added (i.e., PyBench and the Parrot benchmarks), the relative performance of JIT-compiler approaches increases. Therefore, ЯRotor and V8 show how JIT-compilation be used to optimize both kinds of code. This difference with the rest of JIT-based approaches may be due to the fact that ЯRotor and V8 include the intercession primitives in the JIT-compiler, whereas the rest of approaches simulate intercession

with an extra layer.

*4.3. Dynamic Typing*

Apart from structural intercession, we have also added support for dynamic typing to the .NET virtual machine, providing the corresponding services to any .NET language. In order to evaluate the efficiency of these services, we have measured the following four applications that make wide use of duck typing but no structural intercession:

- Pystone. This benchmark is the Python version of the Dhrystone benchmark [56] 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# 4.0 to use the DLR (every reference has been declared as `dynamic`).

- Pybench [55]. A Python benchmark designed to measure the performance of standard Python implementations. Pybench is composed of a collection of 52 tests that measure different aspects of the Python programming language. We have suppressed those tests that use structural intercession (already measured in the previous subsection); those that employ particular features of Python not provided by the other languages (i.e., tuples, dynamic code evaluation, and Python-specific built-in functions); and those that use any input/output interaction. Therefore, 24 tests of the Pybench benchmark were measured in this section.

- Shootout. The third existing benchmark we have used to evaluate our platform is the Shootout benchmark (also known as the Computer Language Benchmarks Game) [57]. This benchmark is composed of different well-known algorithms implemented in both statically and dynamically typed programming languages. We have run those tests that do not perform any I/O interaction, which are: *nbody*, predicts the motion of a group of celestial objects that interact with each other gravitationally; *fannkuch redux*, involves operations (mostly permutations) on vectors of numbers; *spectral norm*, calculates the spectral norm (eigenvalue) of a square matrix using the power method; *mandelbrot*, computes a particular instance of the Mandelbrot fractal set; and *binary trees*, allocates, walks, and deallocates many bottom-up binary trees. As mentioned, we take the C# implementations and translate them into the other languages, ensuring that the same operations are executed in every language.

- Bruckschlegel. A benchmark designed by Thomas Bruckschlegel to evaluate the characteristics of Java, C#, and C++ on Windows and Linux [58]. It is composed of a set of 12 tests that use fundamental data processing and arithmetic operations. Since this last benchmark was designed for statically typed languages, we have removed every type annotation to make it dynamically typed.

Figure 21 shows the average execution time relative to ЯRotor when running these four benchmarks. We can see how ЯRotor is the fastest implementation for short-running applications (start-up). On average, PyPy is the closest implementation, requiring only 1.58% more execution time; V8, the third one, employs 76.81% more time. For long-running applications (steady-state), ЯRotor is the second fastest system after PyPy, which requires 60% the execution time of ЯRotor.

V8, the third fastest implementation, uses 89.55% more execution time than our platform. Most of the tests where ЯRotor shows a low runtime performance are related to (Unicode) string handling. This is because our platform has low runtime performance when dealing with strings. For instance, running the *string concat* test of the Bruckschlegel benchmark, the DLR is 2.49 (start-up) and 3.67 (steady-state) times faster than ЯRotor. This limitation has been inherited from the original SSCLI implementation: as we analyze in the following subsection, where we compare (among others) the CLR and the SSCLI, the SSCLI runs the same test in 16 times more execution time than the CLR [43].

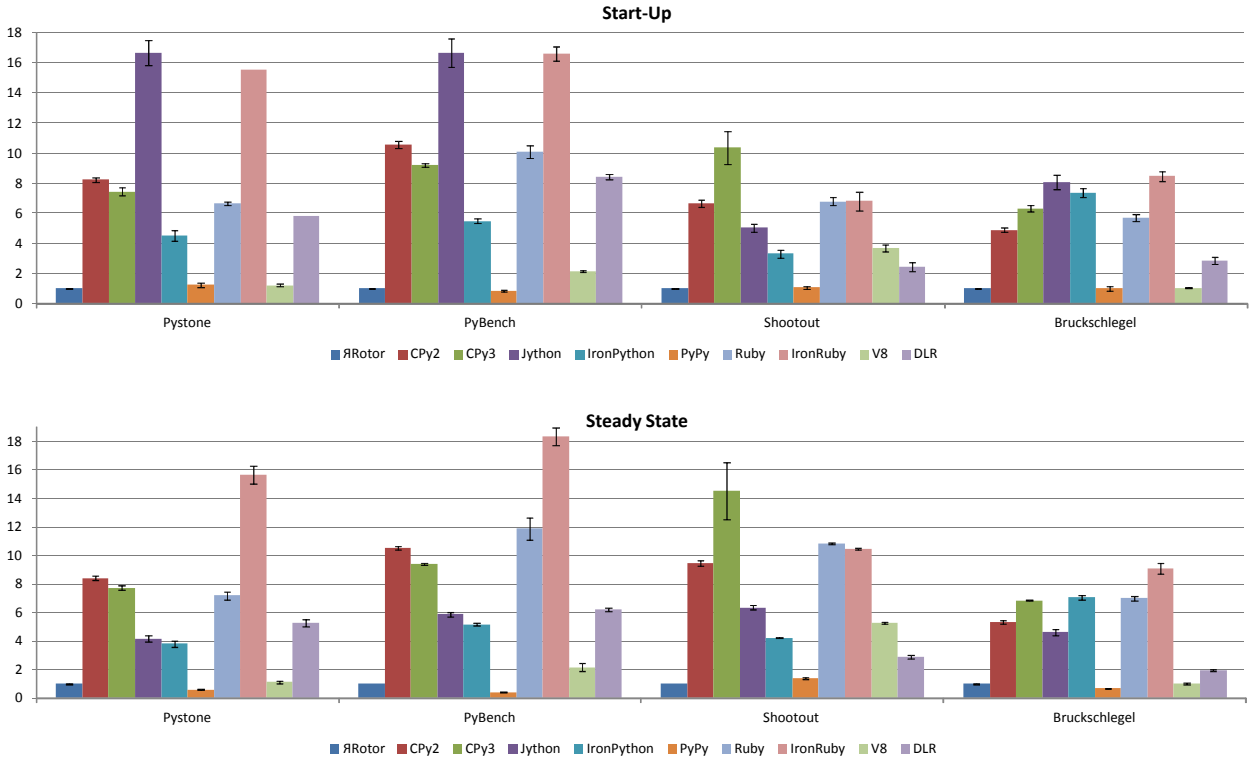


Figure 21: Execution time of the dynamic typing benchmarks relative to ЯRotor.

Regarding memory consumption, ЯRotor shows the lowest memory requirements (Figure 20). Ruby requires 5% more memory than our platform, and the memory consumption of both CPython implementations is 19% higher. Apart from ЯRotor, JIT-compilation implementations use the highest memory resources: 2,052% (Jython), 277% (IronPython), 121% (PyPy), 215% (IronRuby), 38% (V8) and 78.56% (the DLR) more than ЯRotor. Comparing these results with the ones presented in the previous section, it can be observed how JIT-compilation approaches offer better runtime performance when running non-intercessive code, but their relative memory consumption is also increased. Nevertheless, ЯRotor reduces its relative memory resources, showing the best runtime performance in start-up, and the second one in steady-state.

This second evaluation section has shown how adding support of dynamic typing to a JIT-compiler virtual machine provides runtime performance benefits, and it can be implemented with even lower memory resources than interpreted-based approaches. The utilization of JIT-compilation

for implementing the hybrid model shows a higher benefit in these tests compared to intercessive applications. Besides, the evaluation shows that adaptive runtime code optimization and tracing JIT-compilation are two suitable techniques to obtain better runtime performance, especially for long-running applications.

### *4.4. Statically typed code*

In this last group of tests we have measured statically typed applications. The objective of this evaluation is to determine the cost of adding the proposed hybrid model to the SSCLI. For that purpose, we have compared the original SSCLI implementation with ЯRotor, measuring exactly the same applications (written in C#). We have run a set of benchmarks that do not use any feature of the new object model.

We also include the DLR in this evaluation, changing the original C# application (CLR) to another one with all the references declared as `dynamic`. The purpose of measuring execution time of this source code is to assess the current performance cost in the .NET Framework of using dynamically typed references vs. statically typed ones. Therefore, our approach of modifying the underlying virtual machine can be compared with the one that creates an extra layer over the CLR.

We have selected four different benchmarks. First two are those used in the previous section that measure runtime performance of statically typed languages: Shootout and Bruckschlegel. We have also used three real C# applications collected by Ben Zorn [59], and a C# port of a subset of the Java Grande benchmark [60].

The three real applications collected by Ben Zorn consist of a collection of managed code programs 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 computationally 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, the AHCBench size is 1,267 lines computationally intensive code, requiring a relatively small heap.

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

The last benchmark used in this section is a subset of the Java Grande benchmark ported to C# by Chandra Krintz [61]:

- 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 150×150 pixels.

Figure 22 shows the difference between ЯRotor and the SSCLI when none of the new features are used. On average, our platform requires 12% more execution time. Although the new hybrid model added to the platform requires more dynamic checks, its runtime performance is close to the original SSCLI because, when none of the new features have been used, most of the original services are used instead (i.e., many runtime checks are not performed because they are not necessary). ЯRotor consumes 13% more memory than the SSCLI (Figure 23). The execution time of the CLR is 12% (start-up) and 15% (steady-state) of ЯRotor, while this value is only a percentage point higher when compared to the SSCLI. However, the CLR requires 47% more memory than ЯRotor (Figure 23).

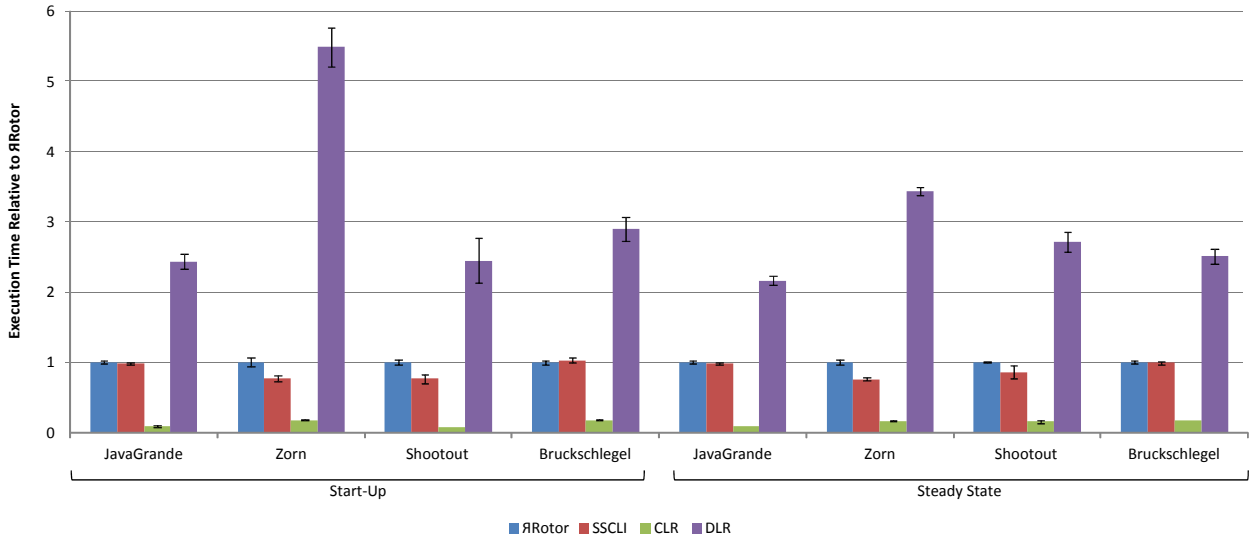


Figure 22: Execution time of the statically-typed benchmarks relative to ЯRotor.

Running this kind of applications, the DLR offers better runtime performance than executing reflective code, requiring 212% (start-up) and 167% (steady-state) more execution time than ЯRotor (Figure 22), and consuming 107% more memory (Figure 23). The DLR performs better than ЯRotor only in 2 tests out of 29 [43]. The first one is the JGFCreate benchmark of the Java Grande suite, which creates loads of objects and assigns them to `dynamic` references. In this test, the DLR performs even better than the CLR (it seems that, for this test, assignments to dynamic references are faster than to statically typed ones): 69% in start-up, and 105% in steady-state. The second program where the DLR is faster is *string concat*, a string intensive test of the Bruckschlegel benchmark. As we have mentioned in the previous subsection, the worse runtime

performance of ЯRotor is an effect of the way the SSCLI implements strings: the DLR is 2.63 and 3.45 times faster than the SSCLI, while this difference rises to more than 16 times comparing the SSCLI with the CLR.

The comparison between the DLR and the CLR shows that the use of `dynamic` involves a performance cost between 1.26 and 1.41 orders of magnitude (19.29 and 27.08 factors) and 41% more memory. These data contrast with the average penalty we have introduced: 12% more execution time and 13% more memory than the SSCLI.

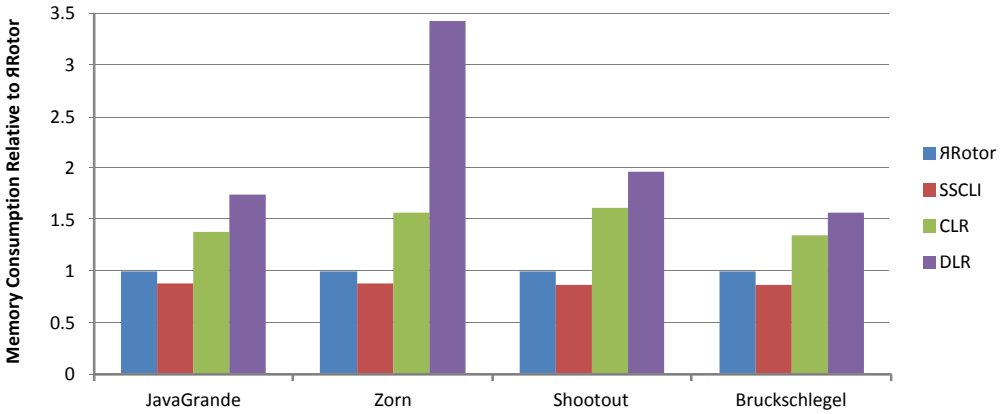


Figure 23: Memory consumption of the statically-typed benchmarks relative to ЯRotor (confidence intervals are not shown because all of them are below 2%).

## 5. Related Work

In this section we analyze the existing work related to ours. We first describe previous work on specifying the semantics of structural intercession. Afterwards, we discuss the virtual machines that provide runtime structural intercession, generating binary code with JIT compilation.

### *5.1. Formal Specification of Structural Intercession*

Abadi and Cardelli defined the ζ-calculus, a pure object-based (prototype-based) calculus where objects are the only computational structures [62]. Although the ζ-calculus is pure prototype-based, it can also be used for defining classes as object generators that are self-contained – including inheritance. An object is a collection of named methods; attributes (fields) are modeled as methods that return a value. Methods can be updated, allowing the modification of attribute values. The update operation produces a new copy of the original object where an existing method is replaced with the new one. However, the ζ-calculus does not provide operations to add and to remove methods from objects because *these operations are harder to handle in a type-theoretical development* [62].

Rémy extended the ζ-calculus with the extension primitive, so that objects could be extended with new methods [63]. The structure of objects was refined to carry more precise information, allowing the definition of width and depth subtyping. This work includes the extension operation of records into the ζ-calculus without including the restriction operation (i.e., removing fields from

a record) [64]. This approach is similar to existing works on changing the class membership of an object while retaining its identity, such as $\mathcal{F}$ickle [22] and *wide classes* [65].

$\delta$ is a simple calculus that provides a formal foundation for an imperative prototype-based system with structural intercession and delegation [66]. They define the operational semantics of reflective languages that support the addition, update and deletion of object methods at runtime –attributes are modeled as methods, following the approach defined by Abadi and Cardelli in their $\zeta$-calculus [62]. Additionally to the list of methods, an object contains a collection of addresses pointing to its parents. When a message sent to an object is not understood, it is automatically forwarded to its parents.

BabyJ is an adaptation of $\delta$ to model a subset of the JavaScript semantics [67]. The objective of BabyJ is to convert object-based weakly typed applications into class-based strongly typed ones. Therefore, they modified $\delta$ to represent a subset of the JavaScript semantics, although the dynamic removal of members was not provided (i.e., full structural intercession was not supported). They also defined BabyJ$^{\mathrm{T}}$ as a typed extension of BabyJ for which a static type system was created. Finally, a semantics preserving transformation of BabyJ$^{\mathrm{T}}$ programs to Java programs provide the generation of class-based applications from prototype-based ones.

EGO is a prototype-based language that offers a type-safe system supporting imperative method addition, removal, and dynamic changes of object inheritance [68]. Its objective is to offer the powerful intercession features of the Self programming language [29], without losing the robustness of a sound static type system. As in $\zeta$-calculus, object members are unified in methods, but their dynamic semantics is more based on $\lambda$-calculus. The type system tracks the linearity of object and method references in order to ensure that objects whose interfaces change are not aliased. They provide a foundation for languages that combine the power of dynamic languages with the benefits of static typing.

### *5.2. Virtual Machines that Provide Structural Intercession*

The Smalltalk virtual machine could be identified as the first example of a widely-known dynamically typed object-oriented reflective virtual machine [69]. It provides structural intercession and duck typing for a class-based object model. The 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 [70] involving important performance improvements (VisualWorks, VisualAge Smalltalk and Digitalk).

Self is a prototype-based object-oriented language for exploratory programming. Unlike Smalltalk, Self uses the prototype-based model to support runtime structural (and partially behavioral) reflection [29]. The prototype-based model allows Self to consistently provide object-level intercession. Its implementation is based on a virtual machine that provides JIT compilation. One of the most important features of Self is the efficient execution of its dynamically typed code [71]. The Self compiler transparently specializes functions for specific argument types based on profiling and gathered statistics (i.e., runtime adaptive optimization).

Parrot is an open source virtual machine designed to efficiently compile and execute bytecode for dynamic languages [72]. The virtual machine is register-based rather than stack-based, and uses continuations as the core means of flow control. Data types in Parrot are defined by means of Polymorphic Containers (PMCs), which model the structure and behavior of each non-built-in

type. The Parrot platform implements two object-oriented PMCs: Object PMC and Class PMC. With these PMCs, Parrot provides a class-based object model that provides structural intercession and duck typing. However, a strong limitation is imposed: these operations throw an exception if the current class has been instantiated [73]. The intercession primitives do not allow changing the structure of a class when the class has any running instance. In order to change this constraint, existing PMCs should be modified.

MetaXa, formerly called MetaJava, is an extension of the Java platform with a reflective meta-level architecture [19]. Structural and behavioral intercession is provided by means of a Meta-Object Protocol (MOP) [74]. The MetaXa approach is quite similar to the one presented in this paper: intercession support added to a production statically-typed class-based virtual machine (integrated into its JIT compiler) to obtain significant performance benefits [75]. The main difference was that MetaXa followed the class-based computational model of the Java programming language. As described in Section 2.1, the class-based object-oriented model of Java does not support object-level reflection in a consistent way [25]. In fact, this model is not the one implemented by most of the dynamically typed object-oriented reflective languages.

The Java Specification Request (JSR) 292 is aimed at supporting dynamically typed languages over the Java platform [12]. Although the JVM has already been used to support dynamic languages such as Groovy or Jython, its runtime performance was not as good as that provided by other implementations (e.g., CPython). A key part of the JSR 292 is the new `invokedynamic` opcode added to the JVM. This instruction has been designed to support the implementation of the message passing mechanism provided by dynamically typed object-oriented languages (duck typing). It provides a dynamic linkage mechanism that helps language implementers to generate bytecode that runs faster in the JVM [76]. The `invokedynamic` specification of the JSR 292 has been included as part of Java 1.7.

The JSR 292 specification also investigates support for *hot-swapping*: the ability to modify the structure of classes at runtime. Although this feature was also expected to be delivered in Java SE 1.7 [12], it was not finally included. However, the Da Vinci Machine (also called the Multi Language Virtual Machine) project [77] has the objective, among others, to provide hot-swapping to the OpenJDK implementation. This project is aimed at prototyping a number of enhancements to the JVM, so that it can run non-Java languages (especially dynamic ones) with a performance level comparable to that of Java itself. This approach is similar to ours in the sense that we extended the semantics of a virtual machine instead of creating a new software layer (as does the DLR). Working at the virtual machine level provides better runtime performance, taking advantage of the JIT-compiler optimizations.

The Dynamic Language Runtime (DLR) [13] is a set of services that run on the top of the CLR, offering a new level of support for dynamic languages on .NET [14]. The DLR is shipped with the .NET Framework 4.0 and it is used to support IronPython, IronRuby, SilverLight, and even C# 4.0, i.e., its new `dynamic` type [36]. Basically, the DLR is a redesign of the object model used in IronPython, allowing any other dynamic language to seamlessly work together, sharing libraries and frameworks. The DLR is a new software layer over the CLR: no modification of the virtual machine was performed to support dynamic languages. It provides duck typing and object-level intercession by means of its `ExpandoObject` class.

The implementation presented in this paper is based on a previous extension of the SSCLI [16].

In that previous work, we described how to include common structural intercession primitives to the SSCLI, obtaining the runtime performance benefits provided by JIT-compilation. Although object-level intercession was supported, the implementation did not provide the hybrid class- and prototype-based model proposed in this paper. The new version allows combining both models in the same application, following the semantic rules described in Section 3. New features such as changing the type of an object at runtime, providing introspective services for the hybrid model, dynamically updating the inheritance tree, and overcoming the inconsistencies between duck typing and dynamic binding described in Section 2.3, have been included. Finally, as mentioned, this new version has been highly refactored and provides new optimizations (Section 3.5).

## 6. Conclusions

This paper proposes a hybrid class- and prototype-based object model to provide the structural intercession and duck typing services of existing reflective languages, supporting the adaptation of any object or class. This model has been implemented as part of a production JIT-compiler virtual machine, obtaining competitive runtime performance and low memory consumption.

We have formalized the semantics of the proposed object model, which allows the dynamic addition, deletion and updating of methods and fields. Any class and object can be adapted, supporting a hybrid class- and prototype-based object-oriented model. Duck typing is also provided to make the most of the new structural intercession services.

We have modified a previous implementation that provides structural intercession in a shared-source version of .NET, including the proposed hybrid class- and prototype-based model. The direct support of the model inside the JIT-compiler virtual machine provides significant performance benefits compared to widely-used approaches. Computing the geometric mean of all the tests, ЯRotor performs 73% (start-up) and 61% (steady-state) better than the second fastest system. Only two language implementations perform better than ЯRotor when running one benchmark. These two systems implement advanced JIT-compilation techniques, such as tracing JIT-compilation and runtime adaptive code optimization, especially suitable for long-running applications. On average, ЯRotor is the JIT-compiler implementation that requires lower memory resources.

Our approach of extending the semantics of the virtual machine performs better and consumes less memory than the existing alternative of creating an extra software layer (the DLR). The average runtime benefits are 435% for intercessive code, 297% when running dynamic typing benchmarks, and 680% for statically typed programs (444% if we compute the geometric mean of these three values). On average, the DLR consumes 63% more memory than ЯRotor. Finally, when none of the new features are used, ЯRotor requires 12% more execution time and 13% more memory than the original SSCLI implementation.

Future work will be focused on retargeting the existing implementation of the *StaDyn* programming language [78] to use ЯRotor as a new back-end. *StaDyn* is a research programming language that supports both static and dynamic typing, extending the semantics of C# [79]. The current implementation generates CLR code, making use of the introspective services offered by the .NET Framework [80]. Future versions will provide structural intercession, generating both

ЯRotor and DLR code. The *StaDyn* type system [80] will provide both static and dynamic typing for the semantics proposed in this paper, and we plan to proof its soundness.

We are also planning to use ЯRotor as a new back-end for the DSAW dynamic aspect-weaving platform [81]. Structural reflection can be used to obtain flexible dynamic aspect-oriented services [82], and we think ЯRotor may involve a notable runtime performance improvement.

The PLT Redex model implementation, the manual and random tests, the source code of the ЯRotor virtual machine, a binary executable version for the Windows platform, and all the examples and benchmarks used in this paper are freely available at:

`http://www.reflection.uniovi.es/rrotor/download/2012/ist`

## Acknowledgments

We would like to thank the anonymous reviewers for their detailed lists of indications, corrections and suggestions that have helped us to improve the article. We also thank Dr. Jay Ligatti for all his comments and feedback that helped to improve this paper.

This work was partially funded by Microsoft Research to develop the project entitled *Extending Dynamic Features of the SSCLI*, awarded in the *Phoenix and SSCLI, Compilation and Managed Execution* Request for Proposals. This work was also funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978, entitled *Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages*.

## References

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

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

[3] A. Hunt, D. Thomas, The Pragmatic Programmer: from Journeyman to Master, Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1999.

[4] ECMA-357, ECMAScript for XML (E4X) Specification, 2nd edition, European Computer Manufacturers Association, Geneva, Switzerland, 2005.

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

[6] G. van Rossum, L. Fred, J. Drake, The Python Language Reference Manual, Network Theory, United Kingdom, 2003.

[7] A. Latteier, M. Pelletier, C. McDonough, P. Sabaini, The Zope2 book, `http://docs.zope.org/zope2/zope2book` (2012).

[8] W. Vanderperren, D. Suvee, Optimizing JAsCo dynamic AOP through HotSwap and Jutta, in: Dynamic Aspects Workshop, 2004, pp. 120–134.

[9] M. Dahm, Byte code engineering, in: In Java-Information's Tage, Springer-Verlag, 1999, pp. 267–277.

[10] S. Chiba, Load-time structural reflection in Java, in: European Conference on Object-Oriented Programming (ECOOP), 2000, pp. 313–336.

[11] M. Grogan, JSR 223. Scripting for the Java platform, `http://www.jcp.org/en/jsr/detail?id=223` (2012).

[12] Oracle, JSR 292, supporting dynamically typed languages on the Java platform, `http://www.jcp.org/en/jsr/detail?id=292` (2012).

[13] B. Chiles, A. Turner, Dynamic Language Runtime, `http://www.codeplex.com/Download?ProjectName=dlr&DownloadId=127512` (2012).

[14] J. Hugunin, Just glue it! Ruby and the DLR in Silverlight, in: MIX'2007, 2007.

[15] A. H. Borning, Classes versus prototypes in object-oriented languages, in: Proceedings of the ACM/IEEE Fall Joint Computer Conference, 1986, pp. 36–40.
[16] F. Ortin, J. M. Redondo, J. B. G. Perez-Schofield, Efficient virtual machine support of runtime structural reflection, Science of Computer Programming 74 (2009) 836–860.
[17] P. Maes, Computational Reflection, Ph.D. thesis, Laboratory for Artificial Intelligence, Vrije Universiteit (1987).
[18] G. Kniesel, T. Rho, S. Hanenberg, Evolvable Pattern Implementations Need Generic Aspects, in: W. Cazzola, S. Chiba, G. Saake (Eds.), ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE), 2004, pp. 111–126.
[19] J. Kleinoder, M. Golm, MetaJava: an efficient run-time meta architecture for Java, Object-Orientation in Operating Systems, International Workshop on (1996) 54.
[20] A. H. Skarra, S. B. Zdonik, Type evolution in an object-oriented database, in: B. Shriver (Ed.), Research directions in object-oriented programming, MIT Press, Cambridge, MA, USA, 1987, pp. 393–416.
[21] 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, pp. 241–258.
[22] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, P. Gianini, Fickle: Dynamic object re-classification, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2001, pp. 120–149.
[23] 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) 69–82.
[24] J. F. Roddick, A survey of schema versioning issues for database systems, Information and Software Technology 37 (7) (1995) 383–393.
[25] J. Kleinöder, M. Golm, MetaJava - a platform for adaptable operating-system mechanisms, in: Proceedings of the Workshops on Object-Oriented Technology, ECOOP '97, Springer-Verlag, London, UK, UK, 1998, pp. 507–514.
[26] D. Ungar, G. Chambers, B. W. Chang, U. Holzl, Organizing programs without classes, in: Lisp and Symbolic Computation, Kluwer Academic Publishers, 1991, pp. 223–242.
[27] S. Ducasse, O. Nierstrasz, N. Scharli, R. Wuyts, A. P. Black, Traits: A mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems (2006) 331–388.
[28] H. Lieberman, Using prototypical objects to implement shared behavior in object-oriented systems, in: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPLSA, ACM, New York, NY, USA, 1986, pp. 214–223.
[29] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 1987, pp. 227–242.
[30] P. Mulet, P. Cointe, Definition of a reflective kernel for a prototype-based language, in: Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software, Springer-Verlag, London, UK, UK, 1993, pp. 128–144.
[31] R. Ierusalimschy, L. H. de Figueiredo, W. C. Filho, Lua – an extensible extension language, Software Practice & Experience 26 (1996) 635–652.
[32] J. B. García Perez-Schofield, E. García Roselló, F. Ortin, M. Pérez Cota, Visual Zero: A persistent and interactive object-oriented programming environment, Journal of Visual Languages and Computing 19 (3) (2008) 380–398.
[33] G. Bracha, Pluggable Type Systems, in: Proceedings of the OOPSLA 2004 Workshop on Revival of Dynamic Languages, ACM, Vancouver, Canada, 2004.
[34] R. Chugh, D. Herman, R. Jhala, Dependent types for JavaScript, in: G. T. Leavens, M. B. Dwyer (Eds.), Object-Oriented Programming Systems and Applications, OOPSLA'12, ACM, pp. 587–606.
[35] J. G. Siek, W. Taha, Gradual typing for objects, in: European Conference on Object-Oriented Programming, ECOOP'07, 2007, pp. 2–27.
[36] G. M. Bierman, E. Meijer, M. Torgersen, Adding dynamic types to C$^{\#}$, in: European Conference on Object-Oriented Programming (ECOOP), 2010, pp. 76–100.
[37] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.
[38] M. Felleisen, R. Hieb, The revised report on the syntactic theories of sequential control and state, Theoretical

Computer Science 103 (2) (1992) 235–271.
[39] G. Klein, T. Nipkow, A machine-checked model for a Java-like language, virtual machine, and compiler, ACM Transactions on Programming Languages and Systems 28 (4) (2006) 619–695.
[40] J. M. Redondo, F. Ortin, Efficient support of dynamic inheritance for class- and prototype-based languages, Journal of Systems and Software 86 (2) (2013) 278–301.
[41] A. Taivalsaari, Delegation versus concatenation or cloning is inheritance too, ACM SIGPLAN OOPS Messenger 6 (1994) 20–49.
[42] J. Banerjee, W. Kim, H.-J. Kim, H. F. Korth, Semantics and implementation of schema evolution in object-oriented databases, in: ACM SIGMOD International Conference on Management of Data, SIGMOD '87, ACM, New York, NY, USA, 1987, pp. 311–322.
[43] F. Ortin, M. A. Labrador, J. M. Redondo, The ЯRotor Structural Reflective Platform, Technical Report. Computer Science Department, University of Oviedo, http://www.reflection.uniovi.es/rrotor/download/2012/tech.report.pdf (2013).
[44] M. Felleisen, R. B. Findler, M. Flatt, Semantics Engineering with PLT Redex, 1st Edition, The MIT Press, 2009.
[45] E. Meijer, J. Gough, Technical Overview of the Common Language Runtime, Microsoft Research, 2001. URL http://research.microsoft.com/en-us/um/people/emeijer/Papers/CLR.pdf
[46] J. Singer, JVM versus CLR: a comparative study, in: Proceedings of the 2nd international conference on Principles and practice of programming in Java, PPPJ '03, Computer Science Press, Inc., New York, NY, USA, 2003, pp. 167–169.
[47] Mono-Project, The Mono project, http://www.mono-project.com (2012).
[48] DotGNU, The DotGNU project: GNU freedom for the net, http://www.dotgnu.org (2012).
[49] C. F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, in: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS '09, ACM, New York, NY, USA, 2009, pp. 18–25.
[50] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: Object-Oriented Programming Systems and Applications, OOPSLA '07, ACM, New York, NY, USA, 2007, pp. 57–76.
[51] D. J. Lilja, Measuring computer performance: a practitioner's guide, Cambridge University Press, New York, NY, USA, 2000.
[52] MicrosoftTechnet, Windows server techcenter: Windows performance monitor, http://technet.microsoft.com/en-us/library/cc749249.aspx (2012).
[53] Microsoft, Windows management instrumentation, http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582\%28v=vs.85\%29.aspx (2012).
[54] G. van Rossum, Parrot benchmark 1.0.4, http://svn.python.org/projects/sandbox/trunk/parrotbench (2012).
[55] P. S. Foundation, Pybench benchmark project trunk page, http://svn.python.org/projects/python/trunk/Tools/pybench/) (2012).
[56] R. P. Weicker, Dhrystone: a synthetic systems programming benchmark, Communications of the ACM 27 (10) (1984) 1013–1030.
[57] Shootout, The computer language benchmarks game homepage, http://shootout.alioth.debian.org (2012).
[58] T. Bruckschlegel, Microbenchmarking C++, C#, and Java, Dr. Dobb's (http://www.ddj.com/cpp/184401976) (2005).
[59] B. Zorn, CLI benchmarks, http://research.microsoft.com/en-us/um/people/zorn/benchmarks (2012).
[60] 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 (2000) 375–388.
[61] C. A. Krintz, A Collection of Phoenix-Compatible C# Benchmarks, http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks (2012).
[62] M. Abadi, L. Cardelli, A Theory of Objects, Springer, New York, 1998.
[63] D. Rémy, From classes to objects via subtyping, in: Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems, ESOP'98, 1998, pp. 200–220.

[64] L. Cardelli, J. C. Mitchell, Operations on records, in: M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, Vol. 442 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 22–52.
[65] M. Serrano, Wide classes, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 1628.
[66] C. Anderson, S. Drossopoulou, δ: an imperative object based calculus, in: International Workshop on Unanticipated Software Evolution (USE), 2002.
[67] C. Anderson, S. Drossopoulou, BabyJ: From object based to class based programming via types, Electronic Notes in Theoretical Computer Science 82 (8) (2003) 53–81.
[68] A. Bejleri, J. Aldrich, K. Bierhoff, L. Pacinotti, Ego: Controlling the power of simplicity, in: Proceedings of the Workshop on Foundations of Object Oriented Languages (FOOL/WOOD), 2006.
[69] A. Goldberg, D. Robson, Smalltalk-80: the language and its implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
[70] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, POPL '84, ACM, New York, NY, USA, 1984, pp. 297–302.
[71] C. Chambers, The design and implementation of the Self compiler, an optimizing compiler for object-oriented programming languages, Ph.D. thesis, Stanford University (1992).
[72] Parrot, Parrot VM homepage, `http://www.parrot.org` (2012).
[73] Parrot, Class PMC implementation, `http://docs.parrot.org/parrot/devel/html/src/pmc/class.pmc.html` (2012).
[74] G. Kiczales, J. D. Rivieres, The Art of the Metaobject Protocol, MIT Press, Cambridge, MA, USA, 1991.
[75] M. Golm, J. Kleinöder, Jumping to the meta level: Behavioral reflection can be fast and flexible, in: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection, Reflection '99, Springer-Verlag, London, UK, UK, 1999, pp. 22–39.
[76] F. Ortin, P. Conde, D. F. Lanvin, R. Izquierdo, Runtime performance of `invokedynamic`: Evaluation through a Java library, IEEE Software 30 (Accepted, to be published) (2013) 1–12. doi:http://doi.ieeecomputersociety.org/10.1109/MS.2013.46.
[77] Oracle, The Da Vinci Machine, a multi-language renaissance for the Java virtual machine architecture, `http://openjdk.java.net/projects/mlvm` (2012).
[78] F. Ortin, D. Zapico, J. Perez-Schofield, M. García, Including both static and dynamic typing in the same programming language, IET Software 4 (4) (2010) 268–282.
[79] F. Ortin, M. García, Union and intersection types to support both dynamic and static typing, Information Processing Letters 111 (6) (2011) 278–286.
[80] F. Ortin, Type inference to optimize a hybrid statically and dynamically typed language, The Computer Journal 54 (11) (2011) 1901–1924.
[81] F. Ortin, L. Vinuesa, J. M. Felix, The DSAW aspect-oriented software development platform, International Journal of Software Engineering and Knowledge Engineering 21 (7) (2011) 891–929.
[82] F. Ortin, J. M. Cueva, Dynamic adaptation of application aspects, Journal of Systems and Software (2004) 229–243.