Efficient Support of Dynamic Inheritance for Class- and Prototype-based Languages

Jose Manuel Redondo, Francisco Ortin

University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain

Abstract

Dynamically typed languages are becoming increasingly popular for different software development scenarios where runtime adaptability is important. Therefore, existing class-based platforms such as Java and .NET have been gradually incorporating dynamic features to support the execution of these languages. The implementations of dynamic languages on these platforms commonly generate an extra layer of software over the virtual machine, which reproduces the reflective prototype-based object model provided by most dynamic languages. Simulating this model frequently involves a runtime performance penalty, and makes the interoperation between class- and prototype-based languages difficult.

Instead of simulating the reflective model of dynamic languages, our approach has been to extend the object-model of an efficient class-based virtual machine with prototype-based semantics, so that it can directly support both kinds of languages. Consequently, we obtain the runtime performance improvement of using the virtual machine JIT compiler, while a direct interoperation between languages compiled to our platform is also possible. In this paper, we formalize dynamic inheritance for both class- and prototype-based languages, and implement it as an extension of an efficient virtual machine that performs JIT compilation. We also present an extensive evaluation of the runtime performance and memory consumption of the programming language implementations that provide dynamic inheritance, including ours.

Keywords: Dynamic inheritance, prototype-based object-oriented model, delegation, dynamic languages, structural intercession, reflection, JIT compilation, SSCLI, virtual machine
2000 MSC: 68-04

1. Introduction

Dynamically typed programming languages have turned out to be suitable for specific scenarios such as Web development, application frameworks, game scripting, interactive programming, rapid prototyping, dynamic aspect-oriented programming, and any kind of runtime adaptable or

Email addresses: redondojose@uniovi.es (Jose Manuel Redondo), ortin@uniovi.es (Francisco Ortin)
URL: http://www.di.uniovi.es/~redondojose/ (Jose Manuel Redondo), http://www.di.uniovi.es/~ortin (Francisco Ortin)

adaptive software. The main benefit of these languages is the simplicity they offer for modeling the dynamicity that is sometimes required to build highly context-dependent software. Common features of dynamic languages are meta-programming, reflection, dynamic inheritance, mobility, and dynamic reconfiguration and distribution.

For example, in Web engineering, Ruby [1] has been successfully used together with the Ruby on Rails framework for creating 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. Currently, JavaScript [4] is widely used 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 [8]. Due to its portability, small size, and ease of integration, Lua [9] has gained great popularity for extending games [10]. Finally, a wide range of dynamic aspect-oriented tools have been built over dynamic languages [11, 12, 1, 13], offering a greater runtime adaptiveness than the common static ones.

The benefits offered by dynamically typed programming languages have caused the addition of their functionalities in some existing statically typed platforms. For instance, the .NET platform was initially released with introspective and low-level dynamic code generation services. Version 2.0 included dynamic methods and the `CodeDom` namespace for modeling (and generating) the structure of high-level source code documents. The *Dynamic Language Runtime* (DLR) adds an extra layer to the .NET platform, providing a set of services to facilitate the implementation of dynamic languages [14]. Microsoft has also included a `dynamic` type to C# 4.0, providing dynamic *duck* typing (passing a message to an object without knowing its static type). `ExpandoObjects`, together with the `dynamic` type, provide structural intercession. The DLR is released as part of the .NET Framework 4.0.

Java has also followed this trend. The last addition to support the features of dynamic languages was the Java Specification Request (JSR) 292 [15], partially included in Java 1.7. The JSR 292 incorporates the new `invokedynamic` opcode in the Java Virtual Machine (JVM) in order to support the *duck* typing features of dynamic languages. The other feature in the JSR 292 was *hot-swapping*: the ability to modify the structure of classes at runtime. This important feature, provided by many dynamic languages, was not finally included in Java 1.7. Since its implementation requires extending the JVM semantics, Oracle launched the *Da Vinci Machine* project [16]. This project aims 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.

However, the great flexibility of dynamically typed languages is offset by two major drawbacks: there is no early detection of type errors, and usually there is a considerable runtime performance penalty. Statically typed languages offer the programmer the detection of type errors at compile time, making it possible to fix them immediately rather than discovering them at runtime—when the programmer's efforts might be aimed at some other task, or even after the program has been deployed [17]. Moreover, the runtime type inspection and type checking performed by dynamically typed languages commonly involve a significant performance penalty.

Since both approaches offer important benefits, the definition of a common platform that sup-

ports both types of languages using the same object model would combine the best characteristics of class- and prototype-based languages. An important performance improvement could be obtained if a production JIT-based virtual machine (such as Java or .NET) is used to implement this common platform. Moreover, if the extensions introduced in the virtual machine are backward compatible (maintain the previous semantics), existing applications may be executed without any change.

Most of the research which has aimed at supporting reflective dynamic languages over the .NET and Java platforms has been restricted to compilers that generate Java or .NET bytecodes simulating the reflective object-model of these languages. Taking Python as an example, there exist different implementations for the Microsoft .NET platform that simulate Python features (Python for .NET from the Zope Community, IronPython from Microsoft that uses the DLR, and the Python for .NET research project from ActiveState). The existing implementation that uses the JVM (i.e., Jython) also follows this approach.

Instead of creating an extra layer over a statically typed virtual machine, our approach focuses on extending an efficient platform with those primitives required in the execution of dynamic languages. The two operations that neither Java nor .NET provide natively are the reflective primitives of structural intercession (adding, modifying, and removing members of classes and objects) and dynamic inheritance (changing the type of objects and classes, and modifying type hierarchies at runtime). In a previous paper, we added structural intercession to an existing JIT-compiler virtual machine [18]. In this paper, we describe how to include the functionalities of dynamic inheritance.

The contributions of this paper are: 1) a formalization of dynamic inheritance in an object-model that supports both the class- and prototype-based approaches; 2) an implementation of the proposed formalization in a production JIT-compiler virtual machine; and 3) an extensive evaluation of the runtime performance and memory consumption of the existing programming language implementations that provide dynamic inheritance.

The rest of this paper is structured as follows. In the next section, we describe the basis of dynamic inheritance over class- and prototype-based object models. Section 3 presents our reflective platform. Section 4 formalizes the dynamic inheritance primitives over both object-oriented models, and our implementation is presented in Section 5. We assess the runtime performance and memory consumption in Section 6. Section 7 discusses related work. Finally, Section 8 presents the conclusions and future work.

2. Dynamic Inheritance

Dynamic inheritance refers to the ability of a programming language to add, modify, or remove the base classes of another class at runtime. It also refers to the ability to dynamically change the type of any object [19]. Languages that support this feature are able to dynamically change inheritance hierarchies. For example, it is possible to insert new types within a specific part of an inheritance hierarchy to dynamically extend the functionality of a group of classes. Moreover, dynamically changing the type of instances allows the programmer to adapt the state and behavior of objects at runtime. This flexible approach is used to create programs that can better adapt to changing requirements.

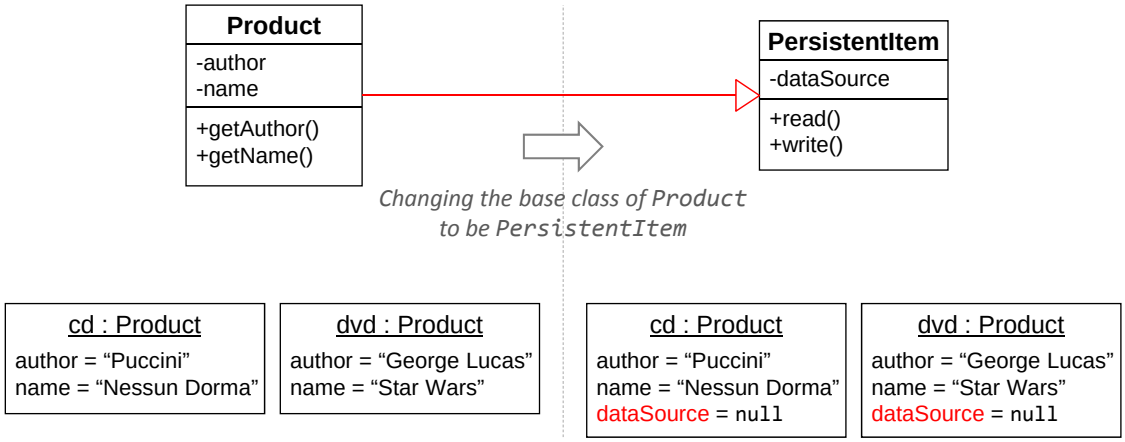

Figure 1: Example of dynamic inheritance in the class-based model.

The semantics of dynamic inheritance depends on the restrictions imposed by the object model implemented by the programming language. The two following subsections explain the two existing approaches to providing dynamic inheritance in class-based and prototype-based object models.

2.1. Dynamic Inheritance in the Class-Based Model

In class-based object-oriented languages, inheritance is defined between classes (as opposed to objects). Java, C#, and C++ are examples of class-based languages. The structure and behavior of an object is defined by its class, which is usually its type. The structure defined in a class must be followed by all its instances (objects) [20]. In the presence of inheritance, the structure of the objects in a class-based language habitually follows a concatenation strategy, which defines the structure of an object as the concatenation of the structure declared in its class and the structure defined by all its superclasses in the inheritance hierarchy [21].

When implementing dynamic inheritance in the class-based model, every instance must have a corresponding class defining all its attributes and methods (state and behavior) [20]. For this reason, any structural change made to the classes has an impact on their instances. This is clearly shown in class-based languages that provide dynamic inheritance [22, 23]: after modifying the structure of a class, the structure of the derived classes and every related instance has to be updated to reflect this change. This way, the concatenation-based inheritance strategy rule is maintained. An example of this behavior is shown in Figure 1. If the `Product` class is changed so as to inherit from `PesistentItem`, the `dataSource` attribute must be added to the `cd` and `dvd` instances and, then, the `read` and `write` methods could be invoked. CLOS [22] and Smalltalk [23] are examples of class-based languages that provide this functionality.

When the class of an instance is changed, those members that are not applicable in the new type should be removed, and those in the new class that were not present in the old one should be added. This mechanism is also referred to as a *type reclassification* in class-based languages: changing the class membership of an object while preserving its identity [24, 25]. It has also been referred to as *schema evolution* in the database world [26]. Examples of class-based languages

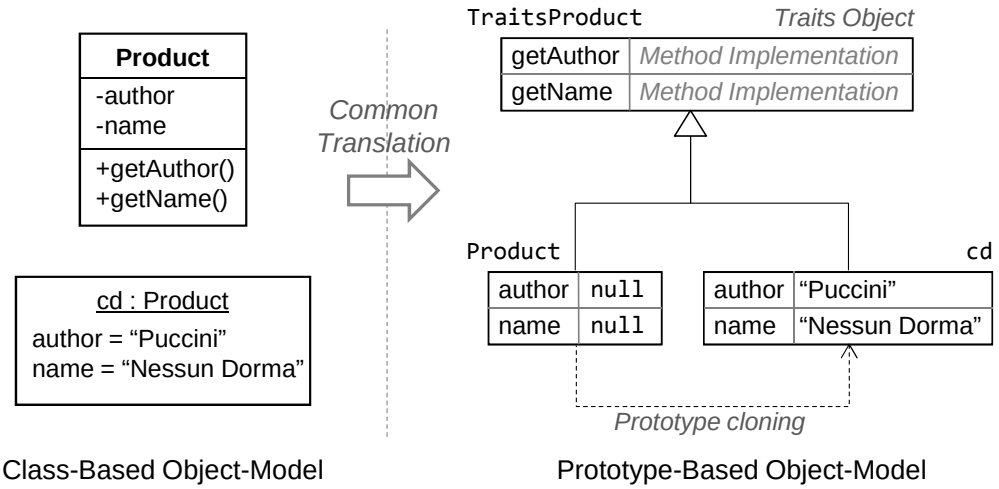

Figure 2: Common translation scheme between the class- and the prototype-based object model.

that allow changing the type of objects are CLOS [22], Smalltalk [27], Bigloo [28] and $\mathcal{F}$ickle$_{\text{II}}$ [25]—all of these will be given in detail in Section 7.2.

2.2. Dynamic Inheritance in the Prototype-Based Model

In the prototype-based object-oriented model, the main abstraction is the object, suppressing the existence of classes [29]. Although this computational model is simpler than the class-based one, any class-based program can be translated into the prototype-based model [30]. This is the reason why it has been previously considered as a universal substrate for object-oriented languages [31, 20].

A common translation from the class-based to the prototype-based object-oriented model is shown in Figure 2. Shared behavior (methods) in a class (`Product`) can be modeled with a *traits* object (`TraitsProduct`) that simply collects the shared methods. The common instance structure defined by classes can be represented with *prototypes* (`Product` object on the right of Figure 2) that define the default state of each object. Object instantiation is performed by cloning a prototype and assigning the specific attribute values to the new instance (`cd`).

In this model, the inheritance relationship is defined between objects (e.g., the relationship between the `cd` and `TraitsProduct` objects in Figure 2). The *is type of* relationship is also modeled with inheritance (e.g., `cd` is a type of product). Method invocation in the prototype-based model is based on *delegation* [32]: when a message is passed to an object, it is checked whether the object has a suitable method or not; in case it exists, it is executed; otherwise, the message is passed to its base object recursively. If the message has not been implemented in the hierarchy, usually an exception is thrown.

The delegation approach facilitates the implementation of both structural changes to objects and dynamic inheritance operations [19]. Replacing any object in an inheritance hierarchy can be done simply by modifying the relationships between the involved objects, without modifying any other entity. Changing the type of an object is also straightforward, because the object only has to be associated with the new “type” modifying the inheritance relationship. Unlike in class-based

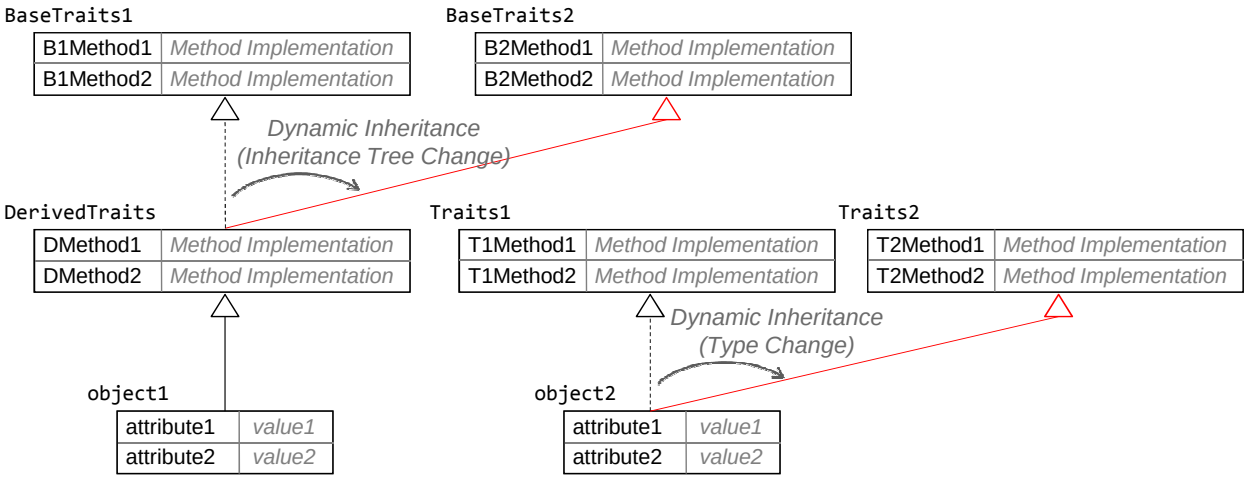

Figure 3: Example scenario of dynamic inheritance in the prototype-based model.

models, attributes are neither added nor deleted. The existing attributes of objects are maintained, because objects are responsible for holding their own set of attributes. The delegation strategy ensures that whenever a member is searched, the most up to date hierarchy structure is used, taking into account the dynamic changes that might have just been made.

The behavior of dynamic inheritance in the prototype-based object-model is described in Figure 3. If the parent of `DerivedTraits` is changed from `BaseTraits1` to `BaseTraits2` (inheritance tree change), then `object1` derived from `DerivedTraits` will be automatically able to respond to the `B2Method1` and `B2Method2` messages. However, it will no longer be able to understand the `B1Method1` and `B1Method2` messages, due to the delegation method search strategy. Likewise, if the base object of `object2` is changed from `Traits1` to `Traits2` (type change), that particular instance will be able to understand the `T2Method1` and `T2Method2` messages, and will no longer be able to respond to `T1Method1` and `T1Method2`. Existing prototype-based languages that provide dynamic inheritance are for example Self [33], Cecil [34], Python [35] and JavaScript [4]—see Section 7.1.

Our research goal is to implement the dynamic inheritance semantics of both class- and prototype-based models as part of a production JIT-compiler virtual machine, which allows efficient support for both kinds of languages. We will first describe how we have added structural intercession to an implementation of the .NET virtual machine (in the next section). Then we will give in detail the design, implementation, and assessment of dynamic inheritance.

3. Adding Structural Intercession to the SSCLI

Reflective Rotor, or ЯRotor [36], is our extended version of the Microsoft SSCLI (*Shared Source Common Language Infrastructure*, also known as Rotor) [37], a shared source implementation of the CLI standard [38]. Microsoft SSCLI is a source code distribution that includes a fully functional implementation of the ECMA-334 C# language standard. This is the only general purpose high-level programming language that is included with this platform. SSCLI also contains the ECMA-335 Common Language Infrastructure specification, various tools, and a set of

libraries suitable for research purposes. ЯRotor offers an efficient implementation of the structural intercession primitives provided by most dynamic languages [18].

3.1. Structural Intercession Primitives

We extended the introspective capabilities of the .NET CLI at the abstract machine level with structural intercession. A new `System.Reflection.Structural` namespace was incorporated in the *Base Class Library* (BCL), supporting two different groups of primitives:

- Field manipulation. Runtime addition, deletion, access and replacement of fields. Besides modifying the structure of a class (modifying the structure of all of its instances), the structure of a single instance can also be modified (object-level reflection).
- Method manipulation. Runtime addition, deletion, invocation, and replacement of methods. The set of messages accepted by an object can be changed at runtime depending on its dynamic context. We provide the functionality of copying methods from one object or class to another, and they can also be dynamically generated by means of the existing `System.Reflection.Emit` namespace. *Duck* typing [1] was included in the semantics of the platform: an object is interchangeable with any other object that implements the same dynamic interface, regardless of whether both objects have a related inheritance hierarchy or not. Duck typing is a widely-used feature provided by the majority of dynamic languages.

All instance and class manipulation primitives were integrated with the new `NativeStructural` class of the BCL. Every class in this library is accessible by all the languages compiled to the virtual machine, so our new features will be available to any language without requiring further changes. In fact, we have not performed any change in the C# compiler to access the new services included in the platform.

3.2. The ЯRotor Computational Model

Since the prototype-based object model supports object-level intercession in a coherent way [18], dynamic languages use this model to provide this level of structural reflection. Therefore, we added the prototype-based model semantics to the virtual machine so that object-level structural intercession could be provided. We maintained the existing class-based model of the original virtual machine to provide backward compatibility. ЯRotor can execute any existing .NET language application previously compiled to the SSCLI. Backward compatibility is provided because our extension does not modify the semantics of the original class-based model.

The resulting modified CLI virtual machine (ЯRotor) is a platform capable of supporting multiple high-level programming languages. Both class- and prototype-based object-oriented models are supported: the former for running statically typed class-based .NET applications; the latter for executing dynamic reflective programs. .NET compilers are responsible for selecting which services of either model are appropriate, depending on the language being compiled and its features. Class-based (e.g., C#) and prototype-based (e.g., Python) languages are directly supported by the virtual machine. Likewise, dynamic (e.g., Ruby) and static typing (e.g., Eiffel) services can also be selected by the compiler. Even hybrid dynamically and statically typed languages (e.g., VB and

Boo) are directly supported by ЯRotor without needing an extra layer to simulate their reflective model (e.g., the DLR).

In conclusion, our reflective extension of the .NET platform supports structural intercession with both class- and prototype-based computational models, implementing static and dynamic typing (the former, performed by the compiler, and the latter, by the virtual machine). This is the foundation from which we have built the support for dynamic inheritance.

4. Dynamic Inheritance Design

We will now formalize dynamic inheritance for the class- and the prototype-based object models supported by our platform. Notice that, while both models are supported, they will not both be present at the same time. Languages could be either class-based or prototype-based, but they will not be both. The language processor is responsible for choosing the appropriate object model.

Dynamic inheritance is provided by a new `setSuper` primitive added to the `NativeStructural` class. This primitive is available to all the languages compiled to the platform. This approach offers the advantage of providing all these functionalities to any language, offering it as a library service rather than as a syntactic extension.

As was described in Section 2, two dynamic inheritance operations are implemented for each object model: Inheritance Tree Change (ITC) and Type Change (TC). These two operations rely on the reflective primitives provided by ЯRotor—see Section 3:

- Inheritance Tree Change. The base type of a class is replaced by another one, performing the necessary changes in the structure of all the classes and instances involved. The changed class, its subclasses, and all their instances, are granted proper access to the members of the new assigned base type. Besides, they cannot access those members that are no longer applicable due to the new hierarchy structure. All changes observe the restrictions of the class-based object model, as described in Sections 2.1 and 2.2.

- Type Change. The type of an instance is replaced by another one, performing the appropriate changes on the structure of the objects. These changes provide access to the members (attributes and methods) exposed by its new type, and forbid access to those members that are no longer applicable. Its effective runtime type is also changed, and it can be obtained through reflection.

We will use the example class diagram in Figure 4 to illustrate the specification of our dynamic inheritance primitives. This diagram represents the entities of a multimedia product factory application. We will take some freedom to change the inheritance relationships in this example to better illustrate all the primitives we have implemented in a simple unique class diagram. A multimedia product is represented by the `Product` class, which stores its `author` and `name` information. `Item` represents a generalization of the `Product` class, allowing products to be managed by other modules without knowing the exact type of product. The `Item` class stores an identifier (`id`) that uniquely identifies a product. Products can be in two different stages during their life cycle:

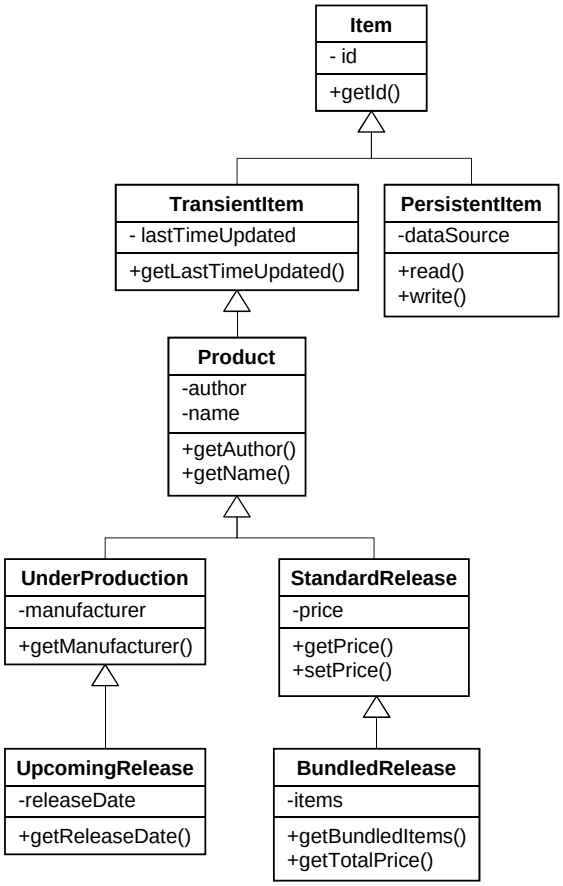

Figure 4: Example class diagram to be adapted by means of dynamic inheritance.

- Unreleased products, which are not on the market yet. They may be under production (`UnderProduction`) or about to be released (`UpcomingRelease`). An `UpcomingRelease` is also an `UnderProduction` product. When a product is under production, it is assigned to a `manufacturer`. When the manufacturer finishes the product, the item is changed to an `UpcomingRelease` state, assigning a new `releaseDate` field to it.

- Released products. There are two types of released products in our example system. A `StandardRelease` is composed of the product, with no extra items. A `price` is assigned to this kind of release so that the release can be sold. The other type of release (`BundledRelease`) represents the releases that package the product with several promotional `items` in order to help sales.

Products can be transient (`TransientItem`), storing the last time the instance has been updated (`lastTimeUpdated`) during the application execution. On the other hand, `PersistentItem` allows existing objects to be stored and read from an external data source (`dataSource`).

Our design describes single dynamic inheritance because the base virtual machine we have used, the SSCLI, does not support multiple inheritance (the same happens in the Java VM). We first define the following sets:

- C_a represents the attribute set of class C.
- C_m represents the method set of class C. The elements in C_m are *mangled* names that combine the method identifier and the type of its parameters, because the virtual machine provides method overloading.
- C_p represents the member set of the class C ($C_p = C_a \cup C_m$).
- The full set of attributes accessible from C is calculated by

$$C_a^+ = C_a \cup D_a^+, \forall D \in baseClassOf(C) \tag{1}$$

- The full set of methods accessible from C is defined by

$$C_m^+ = C_m \cup D_m^+, \forall D \in baseClassOf(C) \tag{2}$$

- Finally, all the members of a particular class C are calculated by

$$C_p^+ = C_a^+ \cup C_m^+ \tag{3}$$

4.1. Class-Based Model

When defining dynamic inheritance in the class-based object model, every instance must have a corresponding class defining all its attributes and methods (states and behaviors). The design of an instance type change is defined as follows.

Suppose X and Y are classes, and that $o : X$ (o is an instance of X). Using the member set defined in (3), the `setSuper`(o,Y) primitive call modifies the structure of o by

1. deleting from o the member set D defined by

$$D = X_p^+ - (X_p^+ \cap {Y_p}^+) \tag{4}$$

2. adding to o the member set A defined by

$$A = Y_p^+ - X_p^+ \tag{5}$$

Figure 5 shows an example of performing a type change in the class-based model—in this figure, and the three following, we include methods in the object diagrams to explicitly describe the messages accepted by each object. It shows the process of releasing a product (changing `cd` from `UpcommingRelease` to `StandardRelease`), adding its selling `price`, and deleting the information that is no longer appropriate to its new state (`manufacturer` and `releaseDate`). In addition, the new `getPrice` and `setPrice` messages are available, whereas the `getManufacturer` and `getReleaseDate` are no longer applicable.

We now specify the inheritance tree change primitive in the class-based model. Suppose X and Y are classes, and let Z be the base class of X. The `setSuper`(X,Y) primitive call modifies the structure of the X class by

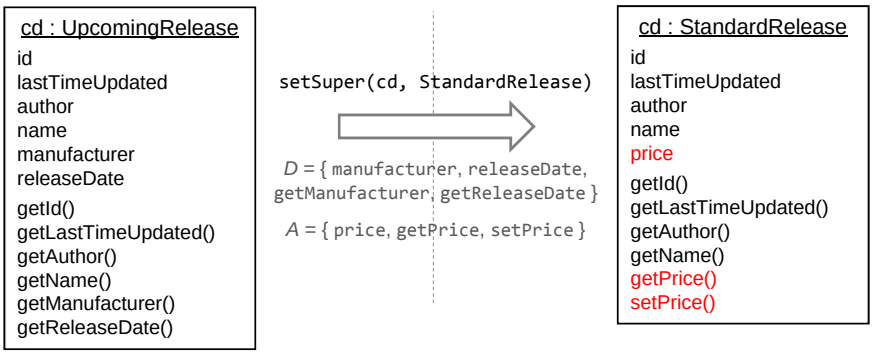

Figure 5: Example type change scenario (class-based model).

1. deleting from X the member set D defined by

$$D = Z_p^+ - (Z_p^+ \cap Y_p^+) \tag{6}$$

2. adding to X the member set A defined by

$$A = Y_p^+ - Z_p^+ \tag{7}$$

Figure 6 shows an example of an inheritance tree change operation in the class-based model. It shows the process of adapting all the products in the system (`cd` and `pack`) to be stored in a persistent storage, adding those members that are necessary to make them persistent (the `dataSource` attribute, plus the `read` and `write` methods). When products are persistent, controlling their last update time (`lastUpdateTime` attribute and `getLastTimeUpdated` method) is no longer needed.

The attributes to be added to an object or class specified by A in (5) and (7) have the default values defined in the ECMA-334 [39] standard specification of the C# language (`0` for numeric values, `null` for object references, and `false` for `bool` values).

As we have seen, dynamic inheritance may involve the dynamic addition and deletion of members of objects and classes. These runtime addition and deletion operations are part of the structural intercession features provided by ЯRotor. When an inheritance relationship is changed, the structure of all the derived classes, and all their instances, must also be changed. As a large number of members can be involved in this operation, members of the modified classes will be dynamically updated only when they are about to be used (lazy adaptation mechanism), improving runtime performance [18].

4.2. Prototype-Based Model

We also provide dynamic inheritance primitives for the prototype-based model. The instance type change primitive is specified as follows. Suppose X and Y are classes, and that $o : X$ (o is an instance of X). Taking the method set defined in (2), the `setSuper`(o, Y) primitive call modifies the structure of o by

1. deleting from o the method set D defined by

$$D = X_m^+ - (X_m^+ \cap Y_m^+) \tag{8}$$

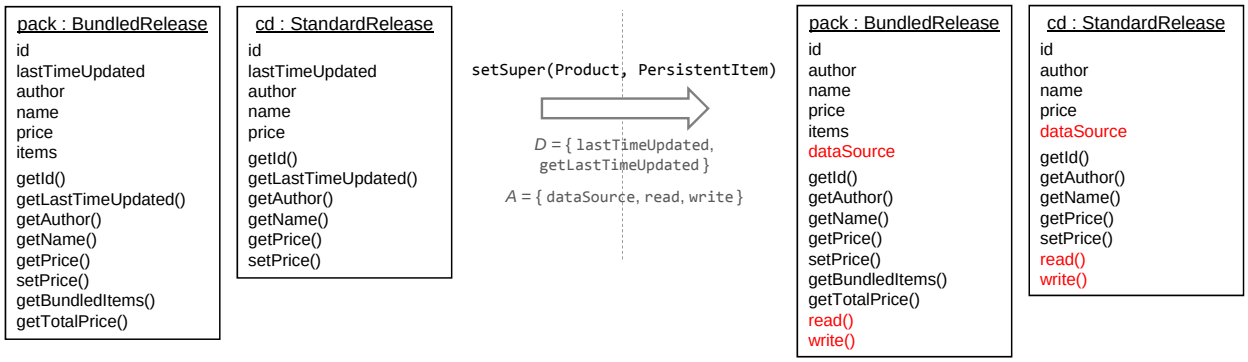

Figure 6: Example inheritance tree change scenario (class-based model).

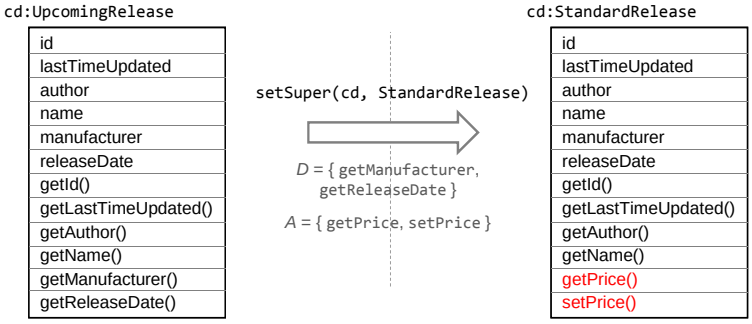

Figure 7: Example type change scenario (prototype-based model).

2. adding to o the method set A defined by

$$A = Y_m^+ - X_m^+ \tag{9}$$

We use the same examples as in the previous section to see the differences between both models. Figure 7 shows an example of type change in the prototype-based model, representing the release of an existing product. We can see how the difference is that, as mentioned in Section 2.2, in the prototype-based model, attributes are neither added nor deleted (the A and D sets only contain methods).

Similarly, we define the inheritance tree change primitive. Suppose X and Y are classes, and let Z be the base class of X. The `setSuper`(X,Y) primitive call modifies the structure of the X class by

1. deleting from X the method set D defined by

$$D = Z_m^+ - (Z_m^+ \cap Y_m^+) \tag{10}$$

2. adding to X the method set A defined by

$$A = Y_m^+ - Z_m^+ \tag{11}$$

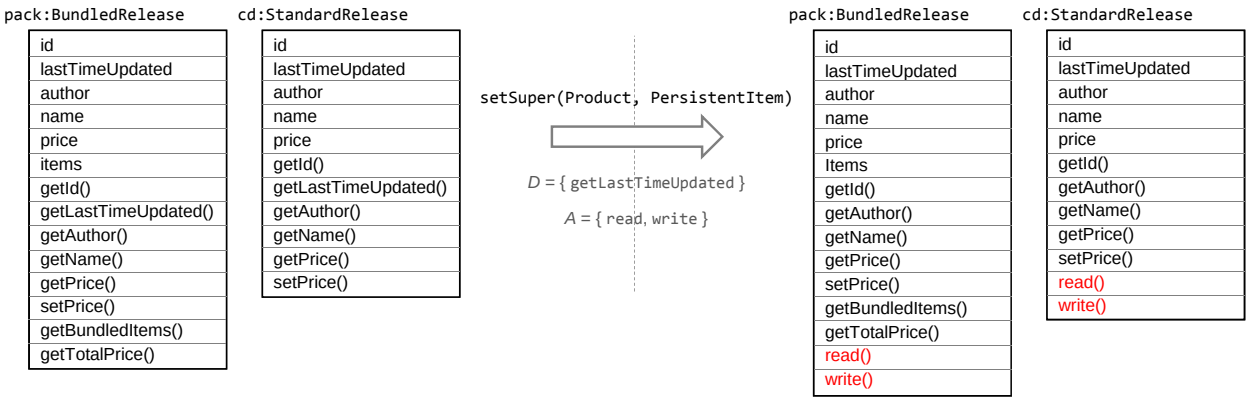

Figure 8: Example inheritance tree change scenario (prototype-based model).

Figure 8 shows the example of inheritance tree change (making the products persistent) in the prototype-based object-oriented model. As in the previous example, only methods are added and deleted, preserving the attributes of each object.

5. Implementation

The implementation of the previously described dynamic inheritance primitives have been made at two levels. The first level (Section 5.1) consists of managed C# code that defines the interface to allow any language to access the new functionalities provided by the underlying virtual machine. Most of the services in this level are simple invocations to native routines implemented in the second level. The second level (Section 5.2) is unmanaged C/C++ code responsible for efficiently implementing dynamic inheritance in the virtual machine internals. In this second level, the JIT compiler, the virtual machine core functions, and the main data structures that support the object model have been modified to achieve our goal. To obtain a high runtime performance, the existing virtual machine services and data structures have been reused whenever possible.

5.1. The `setSuper` Primitive Interface

This is the interface we have defined for the `setSuper` primitive:

- `setSuper(Object, Type, bool)`: Changes the type of the instance referenced by the first parameter to the one described by the second. Classes are represented by the class `Type` in the .NET platform. The `bool` parameter allows choosing between the class-based object-oriented model (if its value is `true`) and the prototype-based one. This way, a language processor of a specific language may choose the appropriate implementation, depending on its object model (Section 4).

- `setSuper(Type, Type, bool)`: Changes the superclass of the class represented by the first parameter to the class described by the second. The third parameter represents the object model (as above).

```
public class DynamicInheritance {

  public Object releaseProductWithPrice(UpcomingRelease product, int price) {
    // Changes the type of product to be StandardRelease (class-based)
    NativeStructural.setSuper(product, typeof(StandardRelease), true);

    // Gets the setPrice method
    MethodInfo method = typeof(StandardRelease).GetMethod("setPrice");
    String sig = NativeStructural.createSignatureFor(method);
    method = (MethodInfo)NativeStructural.getMethod(product, sig);

    // Invokes the method using reflection (duck typing)
    object[] pars = { price };
    NativeStructural.invoke(product, method, method.GetParameters(), pars);
    return product;
  }

  public void storeProducts(Product[] products) {
    // Changes the base type of Product to be PersistenteItem (class-based)
    NativeStructural.setSuper(typeof(Product), typeof(PersistentItem), true);

    // Gets the write method
    MethodInfo method = typeof(PersistentItem).GetMethod("write");

    //Create a suitable mangled name to perform the method search
    String sig = NativeStructural.createSignatureFor(method);
    method = (MethodInfo)NativeStructural.getMethod(typeof(PersistentItem), sig);

    // Writes all the products in the persistence store
    for (int i = 0; i < products.Length; i++)
      NativeStructural.invoke(products[i], method, method.GetParameters(), null);
  }
}
```

Figure 9: Sample C# code illustrating the use of `setSuper` (scenarios in Figures 5 and 6).

Figure 9 shows some sample C# code that makes use of `setSuper` in ЯRotor. The first method, `releaseProductWithPrice`, models the publication of an `UpcomingRelease` product shown in Figure 5: it modifies its type to `StandardRelease` following the class-based model, and then invokes its new `setPrice` method using introspection (note that the static type system does not allow passing that message to `product`). The second method, `storeProducts`, implements the scenario displayed in Figure 6, which makes all products persist: the superclass of `Product` is changed to be `PersistentItem` (in the class-based model), and then the new `write` message is reflectively passed to the `products`.

5.2. Extending the Virtual Machine Internals

The second level of our implementation is C/C++ code that provides dynamic inheritance inside the virtual machine. As we saw in Section 4, implementing dynamic inheritance requires the dynamic addition and deletion of class and object members. We used the `SyncBlock` [37] memory blocks provided by the virtual machine to store these members inside the virtual machine. `SyncBlock` is an internal virtual machine class that was designed to hold additional control structures attached to each individual instance or class in the system. Every object or class in the virtual machine can have a privately owned `SyncBlock` [18].

We used this memory for adding the delegation inheritance strategy of prototype-based languages to the concatenation-based one, originally provided by the SSCLI. If a non-reflective language is being processed, the information is obtained from the original SSCLI data structures. In case dynamic inheritance (or structural intercession) is used, the member information is first consulted in each object's `SyncBlock`, following a delegation strategy [40]. When a message is passed to an object, the method lookup starts by analyzing the current object's `SyncBlock`. If the method is not found, its actual `Type` (class) is checked at runtime. The `SyncBlock` of the class (reflective member set) is analyzed first, and then its `MethodTable` (original member set). If the method is still not found, this algorithm is recursively applied to its superclass. Finally, if the search reaches the `Object` class (there are no more base classes), a `MissingMethodException` is thrown. With this scheme, both computational models are supported. At the same time, dynamic binding semantics is not changed because we start searching from the actual type of the object.

The use of the new dynamic inheritance primitives should involve the adaptation of running programs. However, legacy code does not make explicit calls to the `Reflection.Structural` namespace and, thus, dynamic inheritance changes would not be taken into account within the original program. For instance, a third-party compiled application uses the `callvirt` IL[1] statement to pass a message to an object, instead of using introspection (as does the example code in Figure 9). This is the reason why the dynamic inheritance primitives defined in Section 4 require extending the semantics of some specific IL statements, making existing .NET binary applications adaptable at runtime, without needing to recompile them.

Figure 10 shows an example of some IL code that implements the `releaseProductWithPrice` C# method in Figure 9, but using the new IL semantics instead of introspection (and hence obtaining a better runtime performance). This code might have been generated by a compiler that supports dynamic inheritance and generates code to ЯRotor. It loads the first argument (`product`), the `StandardRelease` class (two next IL statements), the `true` constant (i.e., `1` in IL), and calls `setSuper`. The next paragraph loads the `product` and the `price` arguments and invokes `setPrice`. Notice that, in this invocation, the compiler does not know the type of the `product` object and it uses *duck* typing. For this reason, the `Object` (or any other) class is used to indicate the type where the `setPrice` method is placed. This type is not actually considered by the virtual machine, and a delegation method search strategy is used instead.

In order to extend the semantics of IL, we have modified the native code the JIT compiler generates for some IL statements: `ldfld`, `ldsfld`, `ldflda`, `stfld`, and `stsfld` for attributes, and `call` and `callvirt` for methods. For example, the call and `callvirt` opcodes used in Figure 10 allow executing a method following both the delegation and concatenation inheritance strategies.

The extension of the semantics of IL is performed when the virtual machine JIT-compiler translates IL into x86 native code. A program coded in any .NET high-level programming language is compiled to IL instructions (i.e., `.exe` or `.dll` .NET *assemblies*), not directly to x86 native code. These instructions use names to refer to members instead of offsets (Figure 10). Upon execution, the .NET JIT compiler translates the IL code into executable x86 native code, transforming member names into memory offsets. Our implementation modifies the binary code generated by the JIT compiler, changing the member access to an invocation of a helper function we added to

[1] IL is the Intermediate language of the CLI.

```
.method public hidebysig instance object releaseProductWithPrice(
                            class UpcomingRelease product, int32 price) cil managed  {
    // setSuper(product, typeof(StandardRelease), true)
    ldarg.1
    ldtoken    StandardRelease
    call       class [mscorlib]System.Type [mscorlib]System.Type::GetTypeFromHandle(
                                       valuetype [mscorlib]System.RuntimeTypeHandle)
    ldc.i4.1
    call void [mscorlib]System.Reflection.NativeStructural::setSuper(object,object,bool)

    // Duck typing: Object does not have a setPrice method
    ldarg.1
    ldarg.2
    callvirt   instance void [mscorlib]System.Object::setPrice(int32)

    ldarg.1
    ret
}
```

Figure 10: Example IL code that uses the extended semantics of the virtual machine.

the runtime environment. This function has dynamic access to the up-to-date data in memory, examining the suitable `SyncBlocks` to provide the appropriate behavior. If access to a runtime added member is requested, its memory address is dynamically computed and returned; access to a deleted member causes a runtime exception.

The inclusion of structural reflective information into the `SyncBlock` made us tackle the interaction with garbage collector (GC) operations. The `SyncBlock` is not part of the heap, and hence is not altered by the GC executions. However, dynamically created fields and methods do refer to memory placed in the GC heap, which may be moved or deleted by the GC. In order to solve that problem, the SSCLI manages a data structure (the handle table) that allows data stored in the execution engine memory to safely point to data placed in the GC heap [38]. This way, the hash tables added to the `SyncBlock` store member handles pointing to entries in a handle table, which hold the actual member data.

Method code is not physically removed when methods are deleted by means of intercession or dynamic inheritance. This fact is particularly important when different threads are accessing objects whose members are being erased. Since the SSCLI does not implement a code pitching mechanism (releasing native code from the JIT heap), the binary code of the method is never freed. The handle table not only supports GC interaction but also thread synchronization. The `CreateHandle` function is used when members are about to be added, and the `DeleteHandle` function is used for deleting them. These two functions are synchronized to manage handles in a thread-safe way, supporting the concurrent manipulation of fields.

Since the original virtual machine implements a concatenation inheritance strategy [37], changing the base class pointer of a class representation does not involve adapting either its structure or the structure of its instances. This leaves the system in an inconsistent state, allowing access to members that must not be accessible. Before modifying the base class pointer, the definitions (6), (7), (10) and (11) described in Section 4 are worked out and the class structure is appropriately modified.

In order to help the runtime performance of our dynamic inheritance implementation, we have followed a lazy policy adaptation of object and class structures. The system performs the changes

to objects and classes the first time they are accessed, postponing the adaptation to the moment that it is actually required. The following is the general algorithm implemented to provide this lazy behavior.

1. The method table (type change) or the base class pointer (inheritance tree change) is updated.
2. When the type of an instance is modified, all its members are marked as changed, deferring its adaptation. In the inheritance tree change scenario, the modified types are saved in a list and no instance is marked.
3. When a marked member is accessed, the object adaptation is actually performed. Using delegation, the member is searched for up the hierarchy and the access is finally resolved, unmarking or deleting the member. This search is also carried out when accessing instances of adapted types (inheritance tree change). The first time they are accessed, all their members are marked as changed.
4. If a marked member is not found in a member access, it is then removed from the object or class; otherwise, it is unmarked as changed. Unmarked members are not dynamically searched with delegation, acting as a cache.

6. Performance Evaluation

We have evaluated the runtime performance and memory consumption of the programming languages that provide dynamic inheritance (both class- and prototype-based). This section presents the experimental methodology employed, and the evaluation of the different types of benchmarks.

6.1. Methodology

Three distinct sets of benchmarks have been used to evaluate the efficiency (the runtime performance and memory consumption) of different implementations of dynamic inheritance:

1. Microbenchmark. We have developed a set of synthetic microbenchmarks to measure the efficiency of the dynamic inheritance primitives. We have measured the four possible working scenarios: type and inheritance tree change primitives, using both the class- and prototype-based object models.
2. Existing benchmarks that use dynamic inheritance. We have used several programs that implement the *State* design pattern using dynamic inheritance. This design pattern is a common scenario where dynamic inheritance is applicable. In fact, the *Gang of Four* state that one possible implementation of this pattern is using dynamic inheritance *by changing the object's class at run-time, but this is not possible in most object-oriented programming languages* [41]. These programs combine the use of dynamic inheritance and some other typical computations.
3. The cost of reflection. We have compared the original implementation of the SSCLI 1.0 for Windows with our reflective platform. This has been done to evaluate the performance penalty caused by the introduction of structural intercession and dynamic inheritance. We have used real applications that do not employ any of the new features added to the SSCLI described in this paper. We have also compared the results with the Microsoft CLR 4.0

CLI implementation to estimate what might be the efficiency of our system in case it was included as part of the CLR.

We have selected some dynamically typed object-oriented languages that implement dynamic inheritance following the prototype- and class-based models. The above mentioned benchmarks were ported to the following languages.

- CPython 2.7 and CPython 3.2 [35] for Windows. These are the most widely used Python implementations, developed in C. We have included version 2.7 because, right now, more existing third party software is compatible with Python 2 than with Python 3. The Python programming language implements the prototype-based object-oriented model (classes are actually traits objects; see Section 7.1).

- Jython 2.5.2 [42] (formerly called JPython) over the Java HotSpot Client VM build 1.7 for Windows. This is a 100% pure Java implementation of the Python programming language, seamlessly integrated with the Java platform.

- IronPython 2.7 over the 32-bit CLR 4.0. It is an implementation of the Python language on top of the *Common Language Runtime* (CLR). It compiles Python programs into IL bytecodes that run on either Microsoft's .NET or the Mono open source platform [43]. IronPython 2 has been designed to use the services of the DLR [14]. The DLR is a set of services implemented over the CLR to facilitate dynamic language implementations over the .NET platform.

- Allegro Common Lisp 8.2. This is a mature implementation of the CLOS programming language, available for Windows operating systems [44]. The newer versions have been built focusing on improving their runtime performance [45]. The CLOS language implements the class-based object-oriented model.

- Steel Bank Common Lisp 1.0.37. This is an open-source high-performance Common Lisp compiler [46]. We have used the latest available version for the Windows platform. This implementation offer the best runtime performance among Common Lisp implementations thanks to its native code generation.

- GNU Smalltalk 3.2.3 (GST). This is a free implementation of the Smalltalk-80 language. Other implementations of the Smalltalk programming language, such as VisualWorks Smalltalk [47] and Dolphin Smalltalk [48], have significant restrictions in their dynamic inheritance operations. In these implementations, dynamic inheritance can only be applied to classes that are structurally compatible. Two classes are structurally compatible if they have the same number of members. This is because type change operations reuse existing member slots, so if the source and the destination classes have a different number of members, the operation fails at runtime (`errPrimitiveFailed`). This is the reason why we have not included these implementations in the evaluation. The Smalltalk language implements the class-based object-oriented model.

- Evil Ruby [49] over the Ruby 1.8 language [50]. This is a special Ruby library that extends the Ruby semantics by accessing its internals. It allows the programmer to perform several operations not available in the standard Ruby language [51]. The operations provided include dynamic inheritance. It does not support the Ruby 1.9 language yet. The Ruby language implements the prototype-based object-oriented model (as in Python, classes actually represent traits objects).

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#, combined with IL when dynamic typing is required (as the example code shown in Figure 10).

The selected programs have been translated into the languages mentioned above, ensuring that the values calculated by each program are actually the same. Class hierarchies were also maintained the same, and the most appropriate data structures were selected. We have translated the two dynamic inheritance primitives (type and inheritance tree change) in the following way.

1. Type change: `setSuper(Object, Type)` in ЯRotor; changing the `__bases__` attribute in Python; the `change-class:` message in CLOS; changing the `class` property in Ruby; and the `changeClassTo:` method in Smalltalk.
2. Inheritance tree change: `setSuper(Type, Type)` in ЯRotor; changing the value of the `__class__` attribute in Python; class redefinition in CLOS; the `superclass` property Ruby; and the `superclass:` message in Smalltalk.

All tests were made with code that registers the value of high-precision performance counters that are provided by the Windows operating system [52]. In each test, memory consumption was measured using the `PeakWorkingSet` variable supplied by the Windows Management Instrumentation module.

The measurements were carried out on a lightly loaded 2.67 GHz Intel I7 920 system with 3 GB of RAM running an updated 32-bit version of Windows XP Professional SP3. To evaluate the average percentages, ratios, and multiples, we used the geometric mean.

Regarding the data analysis, we followed the methodology proposed in [53] to evaluate the performance of virtual machines that provide JIT-compilation. We measured the runtime performance once the system reached a steady state, excluding class loading, JIT compilation, and dynamic compilation (of Allegro and SBCL). The methodology was applied by executing each application five times. Each application execution carried out at least ten different iterations of benchmark invocations, measuring each invocation separately. The execution reached a steady state when the coefficient of variation (CoV, defined as the standard deviation divided by the mean) of the last ten iterations fell below the threshold, 2%. We took the mean value of these last ten iterations. A full-heap garbage collection was done before performing each measurement, to reduce the non-determinism across multiple invocations.

6.2. Microbenchmarks

We created four different synthetic microbenchmarks to measure the efficiency of the dynamic inheritance primitives. These microbenchmarks execute type and inheritance tree changes in both

the class- and prototype-based object models. All the tests are based on the example class hierarchy presented in Section 4.

6.2.1. Increasing the Number of Invocations

We first evaluated the efficiency of `setSuper` and the influence of the number of calls on the runtime performance. For this purpose, we measured the execution of the scenarios displayed in Figures 5, 6, 7, and 8. The `setSuper` primitive was used to change types and inheritance trees in prototype- and class-based models. The number of invocations was 1000, 5000, 10,000, and 20,000.

Table 1 shows the execution times in milliseconds of each primitive and tested language. The languages were divided into two groups, depending on their object model. The execution times of the class-based languages are on the left side of the table, and those of the prototype-based languages are on the right. These benchmarks have two implementations in ЯRotor, one using the class-based model (left) and one using the prototype model (right). The first four rows show the assessment of the type change primitive (TC)—the scenario in Figure 5 for the class-based model, and Figure 7 for the prototype-based one. The four last rows show the inheritance tree change (ITC)—the scenario in Figure 6 for the class-based model, and that in Figure 8 for the prototype-based model.

		Class-based Model				**Prototype-based Model**					
	Invocations	**ЯRotor**	**GST**	**ACL**	**SBCL**	**ЯRotor**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
TC	1,000	1.25	1.13	15	31	1.61	5,203	1.01	1.01	10.52	4.76
	5,000	5.07	5	63	125	6.53	26,093	4.24	3.47	25.06	6.97
	10,000	12.20	16	109	265	11	52,218	7.35	7.83	35.68	10.46
	20,000	21.29	27	171	469	22.57	104,593	14.69	14.37	52.31	17.75
ITC	1,000	1.07	1	4,188	5,250	1.01	2,969	1.97	79	7.65	3.41
	5,000	5.4	4	250,048	32,328	4.23	14,766	6.07	390	10.59	7
	10,000	9.78	20	1,038,704	83,531	8.54	29,516	12.08	766	22.21	12.18
	20,000	20.08	34	3,555,363	265,531	14.13	59,062	24.35	1,531	37.68	22.97

Table 1: Execution time (ms) of the microbenchmark, incrementing the number of invocations to `setSuper` (GST stands for GNU Smalltalk, ACL for Allegro Common Lisp, SBCL for Steel Bank Common Lisp, CPy for CPython, Jy for Jython, and IPy for IronPython; TC stands for Type Change and ITC for Inheritance Tree Change).

We can see in Table 1 that a linear increase in the number of invocations involves a linear increase in the execution times. Performing a regression analysis for a linear relationship between the number of invocations and the execution time, the lowest value of the Pearson coefficient, obtained by ACL in the ITC primitive, was 0.97879. The two implementations of Lisp, ACL and SBCL, obtained the highest increase of execution time relative to the increase in the number of invocations. On the other hand, both Jython and IronPython have the lowest such increase. For these two Python implementations, we augmented the number of invocations to 20,000,000, measuring the execution time per invocation. This measurement became stable in both languages (in the range of 100,000 and 500,000 invocations). Jython spends 1.127 (TC) and 1.732 (ITC) microseconds per invocation, whereas IronPython spends 1.004 (TC) and 1.204 (ITC). ЯRotor stabilizes at 0.914 (TC) and 0.706 (ITC) microseconds per invocation.

Figure 11 shows the geometric mean of the runtime performance relative to ЯRotor. When an execution time ratio is much higher than the rest, its representation has been reduced in order

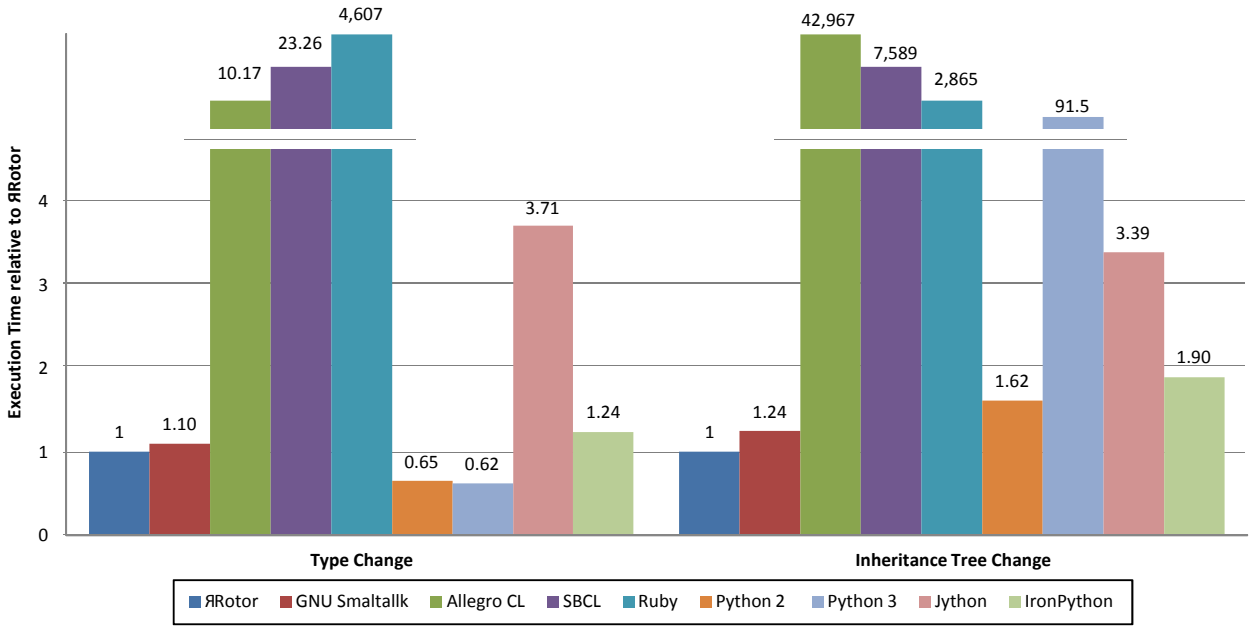

Figure 11: Average execution time relative to ЯRotor, incrementing the number of invocations to `setSuper` (lower values are better).

to improve the visualization of the figure (the values are displayed over each bar). The languages with the higher values are Ruby, Steel Bank Common Lisp (SBCL), and Allegro Common Lisp (ACL), whose execution time was 4607, 23, and 10 times that obtained by ЯRotor for type change, and 2865, 7589 and 42,967 times slower in the case of changing the inheritance tree.

Comparing our system with the two implementations of Python over a virtual machine, i.e., Jython and IronPython, ЯRotor is 2.71 times faster than Jython at executing the type change primitive, and 2.39 for the inheritance tree change. Reflective Rotor was 24% and 90% faster than IronPython. In the case of Smalltalk, our system offers 10% and 24% better runtime performance at running these two primitives.

Only the CPython implementations of the type change functionality had a better performance than ours (53.84% for CPython 2 and 61.29% for CPython 3), whereas ЯRotor offers a faster execution of the inheritance tree change primitive (by 62.84% over CPython 2 and 90 times that of CPython 3). This huge difference between the two versions of CPython in the execution of the second primitive—also observed in the rest of measurements—makes us think that the implementation of this primitive in CPython 3 may not include the optimizations of its previous version yet.

The better runtime performance of CPython in the type change primitive may be due to the specific optimizations that can be implemented when only the prototype-based model is provided. The implementation of type change in this model can be accomplished by merely changing the value of a pointer (i.e., `__class__`). Since ЯRotor provides both object models, the additional work to adapt the instance members to its new type causes this performance difference. As mentioned in Section 5, when ЯRotor performs a type change, all the members in the object are marked as changed in order to postpone the object adaptation to member access. This lazy adaptation policy

		Class-based Model			
	Invocations	**ЯRotor**	**GST**	**ACL**	**SBCL**
TC	1,000	6,647,808	7,888,896	47,566,848	34,258,944
	5,000	7,884,800	7,966,720	48,025,600	37,941,248
	10,000	8,466,432	7,970,816	48,697,344	42,561,536
	20,000	9,555,968	8,101,888	48,697,344	43,159,552
ITC	1,000	7,368,704	7,888,896	46,739,456	46,735,360
	5,000	7,368,704	7,974,912	59,486,208	56,492,032
	10,000	7,368,704	8,040,448	69,750,784	68,845,568
	20,000	7,368,704	8,167,424	72,667,136	96,993,280

		Prototype-based Model					
	Invocations	**ЯRotor**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
TC	1,000	6,656,000	6,025,216	4,636,672	5,816,320	39,526,400	33,157,120
	5,000	7,634,944	11,853,824	4,698,112	5,816,320	39,546,880	33,181,696
	10,000	7,888,896	19,156,992	4,775,936	5,816,320	39,628,800	33,185,792
	20,000	9,527,296	33,746,944	4,886,528	5,816,320	38,653,952	33,198,080
ITC	1,000	7,348,224	5,353,472	4,648,960	5,828,608	38,494,208	33,148,928
	5,000	7,348,224	8,581,120	4,710,400	5,828,608	39,084,032	33,148,928
	10,000	7,348,224	12,623,872	4,792,320	5,828,608	39,501,824	33,177,600
	20,000	7,348,224	20,697,088	4,898,816	5,828,608	39,526,400	33,640,448

Table 2: Memory consumption (KB) of the microbenchmark, incrementing the number of invocations to `setSuper`.

involves a performance optimization when running more realistic workloads (see Section 6.3), but is not as fast as simply changing a pointer (i.e., the CPython approach).

We have also measured the memory consumption. Table 2 shows the values obtained in KBytes. In the type change primitive, all the implementations but CPython 3 increase their memory consumption as the number of invocations increases. For the inheritance tree change, only the memory consumption of ЯRotor and CPython 3 remains constant as the number of invocations increases.

Figure 12 displays the geometric mean of memory consumption relative to ЯRotor. The following languages require several times more memory than ЯRotor in the type and inheritance tree change primitives: ACL (4.98 and 7.31), SBCL (3.87 and 7.79), Jython (4 and 4.33), and IronPython (3.22 and 3.53). Ruby consumes 82% and 42% more memory than our platform, but Smalltalk is almost the same as out platform (1% less for type change, and 9% more in the inheritance tree manipulation). CPython implementations require less memory than ours. They require 60% (CPython 2) and 74% (CPython 3) of the memory we use in type change, and 65% and 79% for the second primitive. This difference may be caused by the common memory consumption introduced by the use of JIT compilation [54]. The performance advantages offered by a JIT compiler are commonly offset by its higher memory consumption [54]. This fact is observed in the four different implementations of Python, where those that use a JIT-compiler (Jython and IronPython) require significantly more memory than the interpreted-based ones (CPython 2 and 3)—this trend is visible in the results for all the benchmarks presented in this paper.

6.2.2. Increasing the Number of Members

After analyzing the effect of the number of invocations of the `setSuper` primitive on the runtime performance, we now analyze the influence of the number of members involved in a dynamic inheritance operation. As explained in Section 4, dynamic inheritance operations are

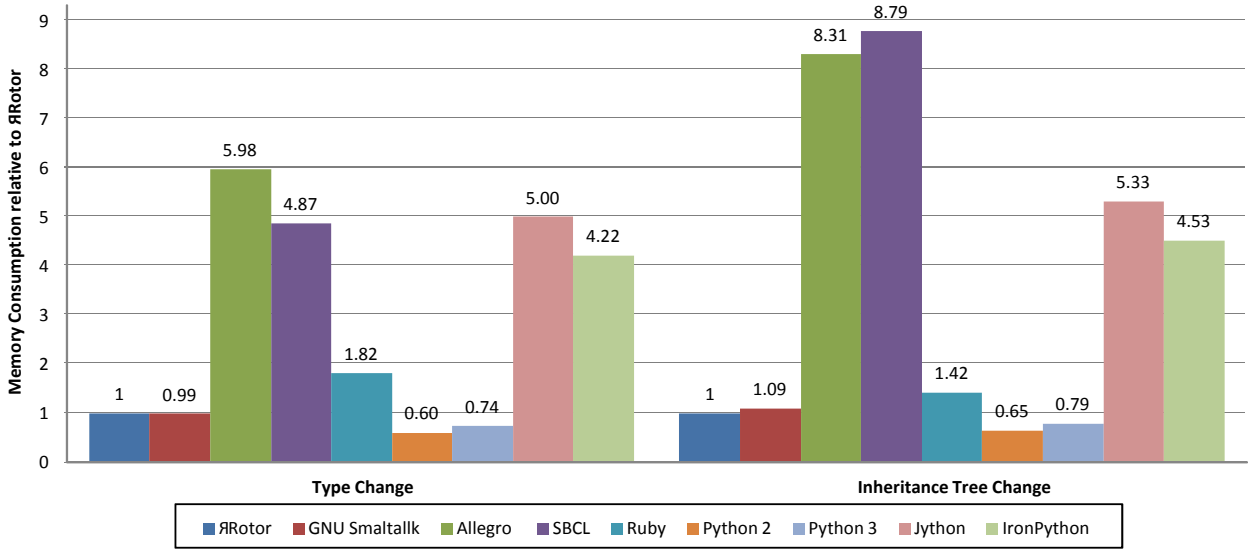

Figure 12: Average memory consumption relative to ЯRotor, increasing the number of invocations to `setSuper` (lower values are better).

composed of the addition and deletion of members to classes and objects. Figure 5 shows that changing the `cd` instance from `UpcomingRelease` to `StandardRelease` involves the addition of the members `manufacturer`, `releaseDate`, `getManufacturer`, and `getReleaseDate`, and the deletion of `price` and `getPrice`. This second microbenchmark increases the number of methods and attributes (10, 100, and 1,000) to be added and deleted (the *A* and *D* sets defined in Section 4) in the invocation of `setSuper`—the number of invocations has been fixed at 5000. The scenarios are those described by Figures 5, 6, 7, and 8.

Table 3 shows the execution times in milliseconds. There are two interesting observations to make regarding the influence of the number of members on the runtime performance. The first issue is that an increase in the number of methods does not seem to involve a notable decrease in the runtime performance in any language. We carried out a regression analysis for a linear relationship between the number of methods and the execution time, conducting an Analysis of Variance (ANOVA). In that analysis, Jython obtained the greatest variable coefficient of execution time, but only 0.0163 for type change and 0.0098 for inheritance tree modification. We think this low dependence is produced by the fact that methods are placed in the representations of classes in both models, and changing the inheritance and type relationships does not produce a modification of the internal structures of classes and objects.

The second issue we have identified from the data in Table 3 is that, unlike prototype-based languages, the runtime performance of class-based languages seems to deteriorate when the number of attributes is increased (ЯRotor is an exception because we provided a hybrid approach). GST, ACL, and SBCL (all class-based languages) show a clear dependence of the execution time on the number of attributes, for both primitives, whereas the execution times of Python and Ruby (prototype-based) have almost no variation. The class-based object model requires the object structure to be modified when dynamic inheritance is used, causing a runtime performance penalty. This penalty does not take place with methods, because it does not involve the dynamic evolution

of classes and objects.

The performance of ЯRotor does not depend significantly on the number of members, even though it is implemented over a class-based virtual machine. This is because of the lazy adaptation strategy we followed in the implementation (Section 5), which is closer to the prototype-based object-model semantics.

			Class-based Model				Prototype-based Model					
		Memb.	**ЯRotor**	**GST**	**ACL**	**SBCL**	**ЯRotor**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
TC	Attrib.	10	6.33	2	125	219	5.89	26,156	3.71	3.66	27.49	7.78
		100	14.55	16	1,562	1,804	14.55	26,171	3.81	3.58	23.46	11.13
		1,000	14.83	1,219	15,574	66,625	16.35	27,953	3.23	3.59	25.99	59.81
	Meth.	10	6.38	5	47	125	6.09	26,375	3.72	3.18	24.22	7.61
		100	6.61	17	47	125	6.71	26,500	3.83	3.4	39.23	7.18
		1,000	6.76	18	47	125	6.6	26,797	3.85	3.65	46	7.92
ITC	Attrib.	10	5.30	16	254,672	35,234	5.47	14,781	5.94	375.00	10.01	7.74
		100	5.58	18	599,102	103,805	5.27	14,906	5.89	375.00	12.80	10.70
		1,000	5.48	1,216	3,880,875	937,406	5.25	15,484	6.14	375.00	12.98	37.80
	Meth.	10	4.99	4	208,547	32,578	4.89	14,797	5.68	375.00	12.99	7.88
		100	5.18	15	296,953	32,140	5.27	15,062	6.28	360.00	19.23	7.19
		1,000	5.71	16	326,648	33,609	4.79	15,219	6.17	437.00	25.00	7.47

Table 3: Execution time (ms) of the microbenchmark, incrementing the number of members.

Figures 13 and 14 display the geometric mean of the execution times relative to ЯRotor obtained from the data in Table 3. The results are very similar to those analyzed in the previous section. ACL, SBCL, and Ruby are many times slower than ЯRotor: ACL is 66 (TC attributes), 6.62 (TC methods), 114,046 (ITC attributes), and 64,767 (ITC methods) times slower; 146 (TC attributes), 19 (TC methods), 17.8 (ITC attributes), and 5.14 (ITC methods) for SBCL; and 2,913 (TC attributes), 4,287 (TC methods), 2,753 (ITC attributes), and 2,815 (ITC methods) for Ruby. Jython is 1.6, 3.99, 1.29, and 2.35 times slower than ЯRotor in the four different scenarios. We are also 41%, 14%, 140%, and 54% faster than IronPython, and 54%, 1%, 347%, and 5% faster than Smalltalk.

As before, the two implementations of CPython are the fastest for the type change primitive. CPython 2 requires only 38% and 60% of the time ЯRotor employs to run the same code, and these percentages are 37% and 56% for CPython 3. However, in the inheritance tree change tests ЯRotor was 19% and 26% faster than CPython 2, and 74 and 80 times faster than CPython 3. As in the previous section, the runtime performances of these two implementations of Python differed notably when executing this dynamic inheritance primitive.

The memory consumption of this second microbenchmark is shown in Table 4. Figure 15 shows the geometric mean of the memory consumption relative to ЯRotor for each language and operation. The results are similar to those obtained in the previous memory measurements. The memory consumption of ACL, SBCL, Jython, and IronPython is several times (from 3.1 to 11.57) greater than that of ЯRotor. On average, Ruby and Smalltalk use 31.36% and 7.84%, respectively, more memory than our platform. As in the previous section, CPython implementations are the ones that require less memory: the average consumption is 70.36% and 86.08% of that of ЯRotor.

IronPython has more memory consumption and worse runtime performance than ЯRotor, even though these two are based, respectively, on the CLR and SSCLI implementations of the CLI. IronPython uses the DLR (see Section 7). Even though the CLR is notably faster than the SSCLI—3.34

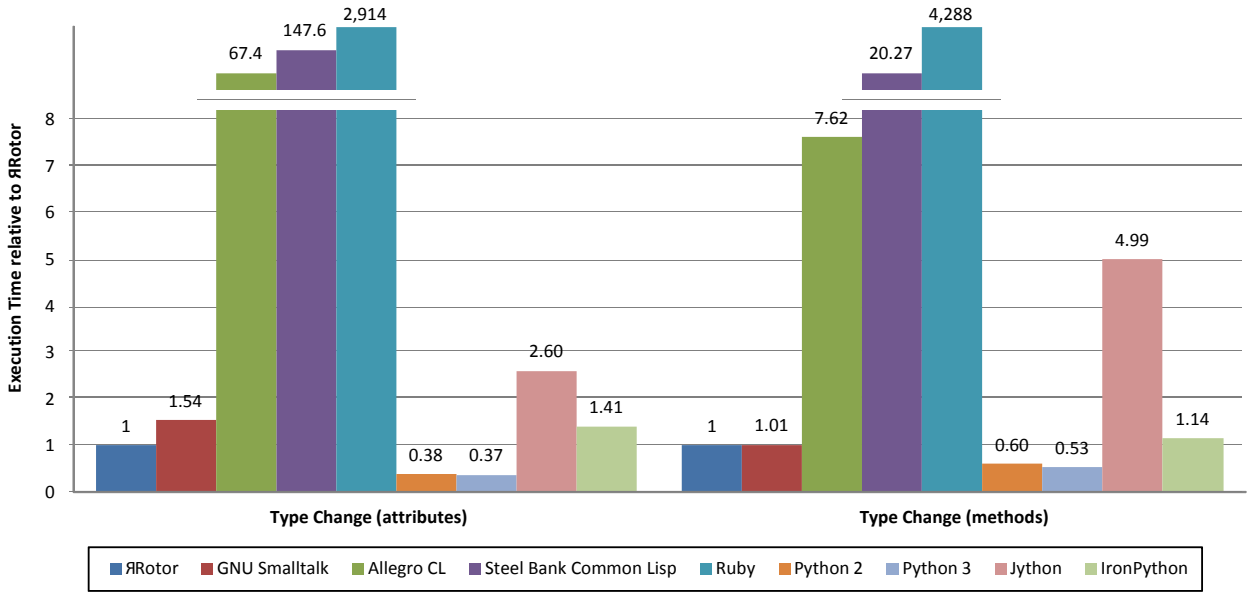

Figure 13: Average execution time relative to ЯRotor, running the type change primitive (lower values are better).

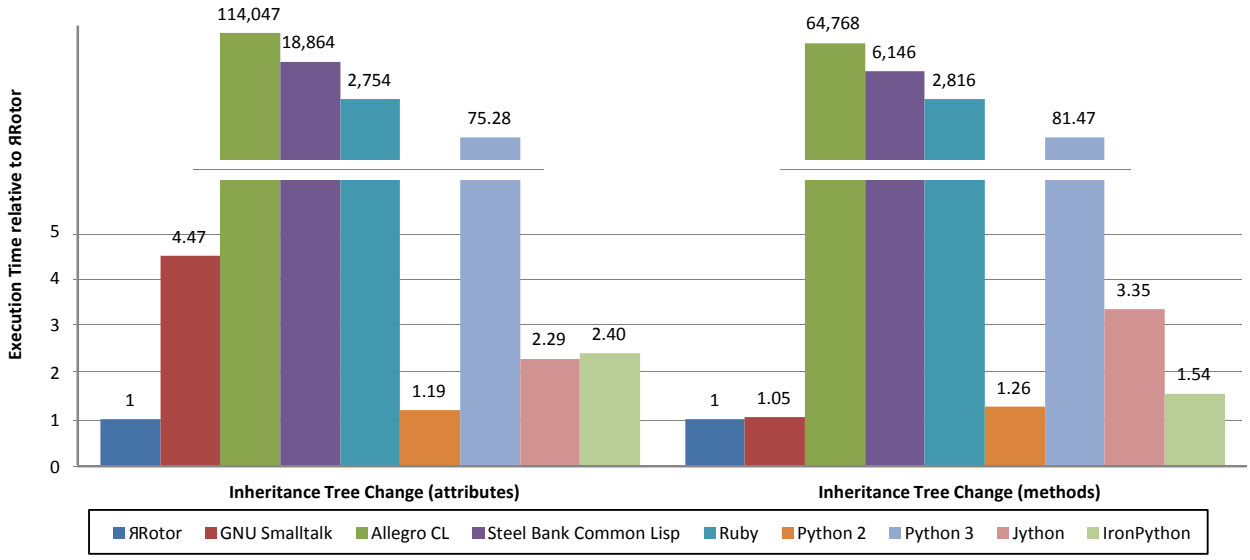

Figure 14: Average execution time relative to ЯRotor, running the inheritance tree change primitive (lower values are better).

times faster in the assessment detailed in Section 6.4—ЯRotor (based on the SSCLI) is significantly more efficient than IronPython (based on the CLR).

6.3. Execution of Existing Benchmarks

The objective of the previous microbenchmarks was to measure the efficiency of the dynamic inheritance primitives, including an analysis of its dependence on the number of invocations and members. Although this first assessment provides basic information about the efficiency of these

		Members	ЯRotor	GST	ACL	SBCL
TC	Attrib.	10	8,105,984	8,245,248	47,136,768	42,676,224
		100	8,278,016	9,998,336	58,019,840	45,342,720
		1,000	8,417,280	9,633,792	156,074,261	61,313,024
	Meth.	10	7,905,280	7,983,104	47,022,080	40,124,416
		100	7,933,952	8,081,408	71,077,888	45,473,792
		1,000	8,413,184	8,560,640	311,635,968	62,521,344
ITC	Attrib.	10	7,348,224	8,249,344	59,486,208	57,511,936
		100	7,348,224	9,957,376	78,864,384	62,443,520
		1,000	7,442,432	9,637,888	291,754,150	132,182,016
	Meth.	10	7,352,320	8,048,640	52,940,800	57,061,376
		100	7,389,184	8,155,136	53,116,928	57,245,696
		1,000	7,827,456	8,568,832	54,878,208	76,152,832

		Members	ЯRotor	Ruby	CPy2	CPy3	Jy	IPy
TC	Attrib.	10	8,105,984	11,964,416	4,694,016	5,820,416	38,977,536	33,206,272
		100	8,278,016	11,759,616	4,919,296	5,980,160	40,493,056	33,169,408
		1,000	8,417,280	13,312,000	6,742,016	8,183,808	43,900,928	34,455,552
	Meth.	10	7,905,280	11,956,224	4,706,304	5,828,608	38,760,448	33,177,600
		100	7,933,952	11,759,616	4,853,760	6,135,808	42,418,176	33,284,096
		1,000	8,413,184	13,266,944	8,118,272	9,555,968	78,995,456	34,476,032
ITC	Attrib.	10	7,348,224	8,572,928	4,710,400	5,836,800	38,690,816	33,189,888
		100	7,352,320	8,433,664	4,923,392	5,996,544	40,415,232	33,189,888
		1,000	7,450,624	9,867,264	6,742,016	8,183,808	44,244,992	34,447,360
	Meth.	10	7,348,224	8,572,928	4,718,592	5,840,896	39,591,936	33,202,176
		100	7,393,280	8,429,568	4,870,144	6,148,096	41,349,120	33,222,656
		1,000	7,827,456	9,818,112	8,118,272	9,555,968	58,920,960	34,422,784

Table 4: Memory consumption (KB) of the microbenchmark, incrementing the number of members.

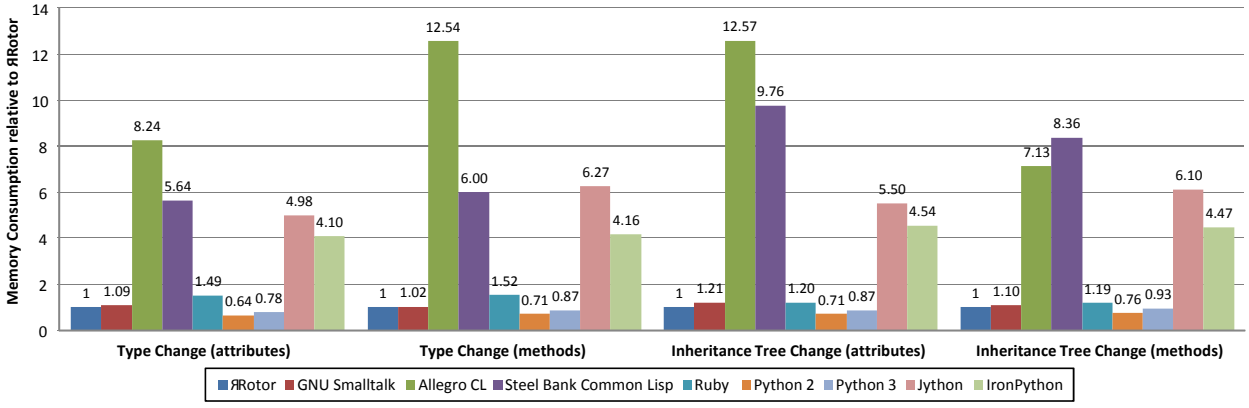

Figure 15: Average memory consumption relative to ЯRotor, increasing the number of members (lower values are better).

primitives, the benchmarks did not consider realistic workloads. As an example, we measured the time required to change the type of an object, but its new members were not accessed afterwards. This is the reason why in this section we now evaluate the efficiency of dynamic inheritance under more realistic workloads. We have translated three existing benchmarks that use dynamic inheritance into the selected languages, evaluating their runtime performance and memory consumption.

	Class-based Model			
	ЯRotor	**GST**	**ACL**	**SBCL**
Execution Time	172	421	36,203	16,562
Memory Consumption	10,485,760	12,353,536	47,439,872	71,847,936

	Prototype-based Model					
	ЯRotor	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
Execution Time	172	371,437	421	36,203	16,562	371,437
Memory Consumption	10,485,760	107,589,632	12,353,536	47,439,872	71,847,936	107,589,632

Table 5: Execution time (ms) and memory consumption (KB) of the Word Count application.

6.3.1. Word Count

Word Count [55] is the first application of the set of three existing benchmarks we have used to evaluate dynamic inheritance. Words, lines, and characters within a text are counted, examining each character individually. This program implements the *State* design pattern [41] using dynamic inheritance. An instance changes its type between classes that represent all the possible states of an automaton. It defines three different states (inside a word, blank space, start of a new line), each one modeled by a different class. The object (automaton) varies its behavior depending on its state. Since the state change is a frequent operation, dynamic inheritance is intensively used at runtime. We used a string with 70,000 characters to run this benchmark.

Table 5 shows the execution times and memory consumption of this program. Figure 16 displays the relative runtime performance, dividing the values in Table 5 by those exhibited by ЯRotor. In this scenario, our platform obtains both the best runtime performance and the lowest memory consumption (the highest efficiency). Smalltalk is the next best, consuming 145% more time and 18% more memory. In this test, the CPython implementations are more than 2.7 times slower, and the memory consumption of CPython 2 is almost the same as ours (although CPython 3 needs 2.49 times more memory). The rest of the systems have a similar pattern as in the previous assessments. IronPython and Jython are 4.27 and 5.91 times slower. The ratios of SBCL, ACL, and Ruby are 96, 210, and 2,161, respectively.

In the execution of a more realistic program, the use of a JIT-compiler seems to provide significant runtime performance benefits with an efficient level of memory consumption. The evaluated program makes frequent use of the type change operation. This primitive also involves the use of *duck typing* because of the dynamic typing nature of dynamic inheritance (the `releaseProductWithPrice` method in Figure 9 is an example of this connection). The runtime performance benefits provided by ЯRotor when duck typing is used [18] also seem to have had a considerable bearing on the comparative performance increase.

6.3.2. Pybench

We now evaluate the second existing benchmark that uses dynamic inheritance. Pybench is a Python benchmark designed to measure the performance of standard Python implementations [56]. It is composed of a collection of 52 tests that measure different aspects of the Python programming language. We have suppressed those tests 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. We have translated 30 tests of the Pybench

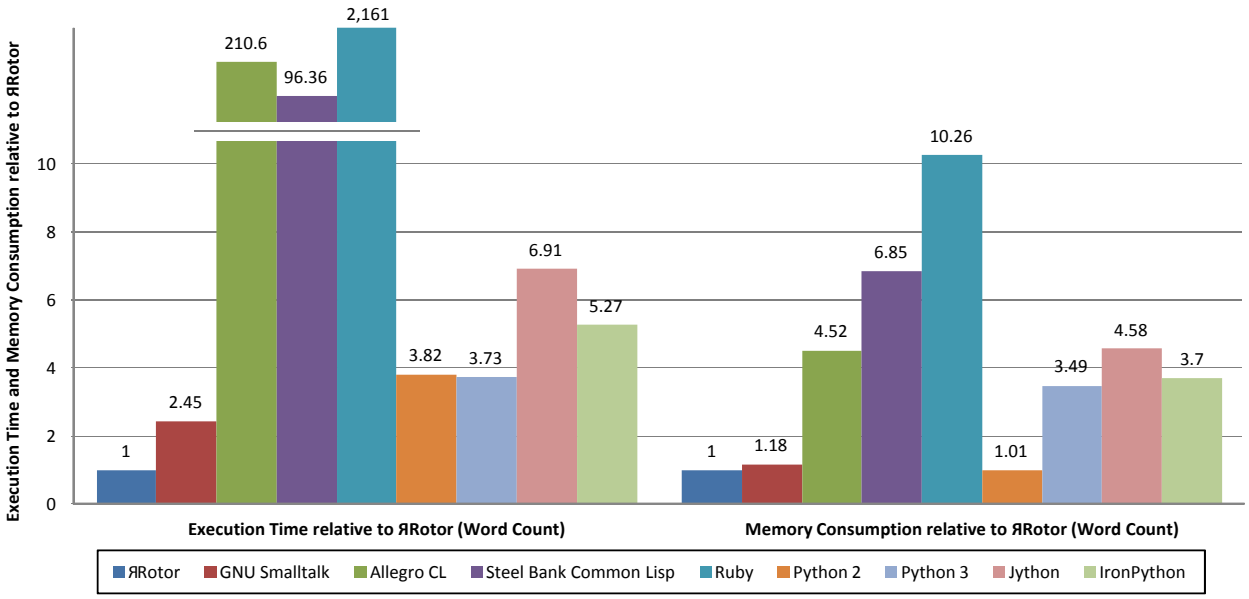

Figure 16: Average execution time and memory consumption relative to ЯRotor, running the Word Count application (lower values are better).

benchmark.

Each test was modeled by a single class that implemented its execution in a `run` method. We created an object whose type changes from one test to another, invoking its corresponding `run` method. The 30 tests were sequentially run using dynamic inheritance. This benchmark makes moderate use of dynamic inheritance and extensive use of dynamic typing operations.

Figure 17 shows the execution times and memory consumption relative to ЯRotor of executing the whole Pybench benchmark. In this scenario, ЯRotor also obtained the best runtime performance, and only the two CPython implementations employed less memory: 87% (version 2) and 97% (version 3). If we calculate efficiency as the ratio of runtime performance (the inverse of execution time) to memory consumption, our platform obtains the best efficiency, being 103% and 158% better than CPython 2 and 3, respectively.

Pybench combines dynamic inheritance with other common operations of dynamic languages. The influence of these non-reflective operations has made IronPython improve its relative runtime performance (between the two CPython implementations). However, IronPython consumes five times more memory than CPython. Jython has also improved its runtime performance over the previous section. These two JIT-compiler VM Python implementations (IronPython and Jython) have improved their runtime performance in this scenario, while requiring more memory resources. On the other hand, Smalltalk has the opposite tendency (it does not implement a JIT-compiler): memory consumption is only 27% higher than ЯRotor, but runtime performance is 45 times worse—it was significantly lower in the assessment of dynamic inheritance.

The source code in Pybench combines dynamic inheritance with other common operations of dynamic languages. The influence of running these operations has given IronPython the better runtime performance of the two CPython implementations, although the memory consumption is significantly higher (more than a factor of five). Since this trend is also observed in Jython,

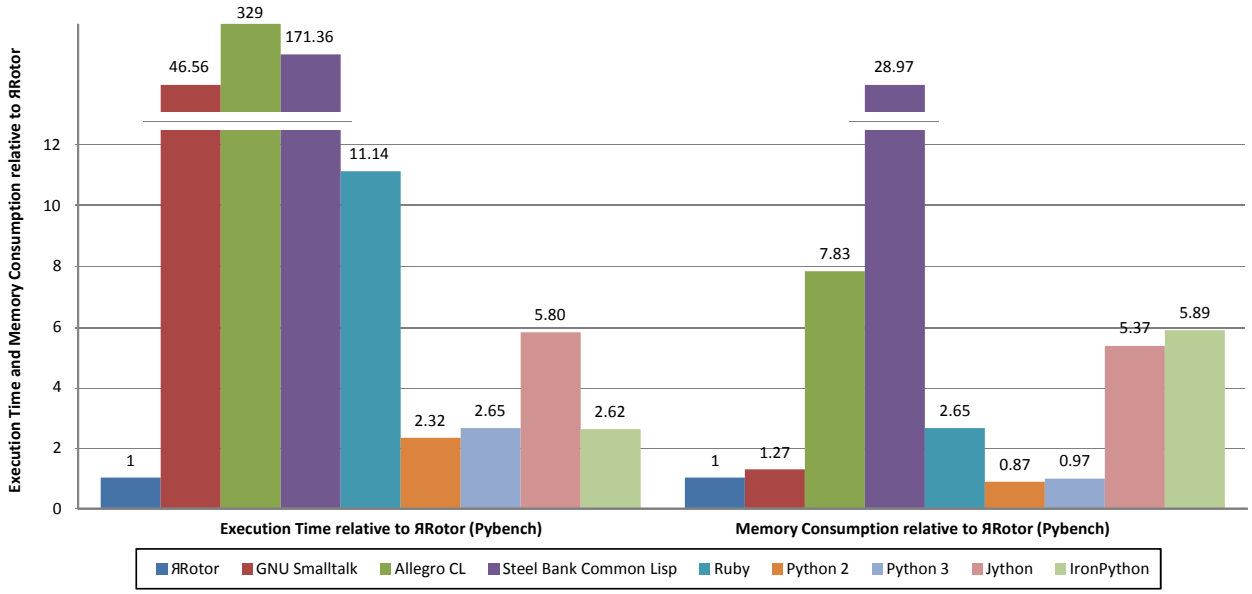

Figure 17: Average execution time and memory consumption relative to ЯRotor, running the Pybench benchmark (lower values are better).

it may be caused by the optimizations performed by Java and .NET JIT-compilers (commonly requiring more memory resources). Smalltalk, however, has the opposite tendency (it does not implement a JIT-compiler): the memory consumption is only 27% higher than ЯRotor, but the runtime performance is 45 times worse—while it was by a factor less than one in the assessment of dynamic inheritance.

Table 6 gives in detail the execution times of each single test in Pybench (the memory consumption of the whole benchmark is shown in Table 7), allowing a more detailed analysis. In many of the tests that perform Unicode string operations (`ConcatUnicode`, `CreateUnicodeWithConcat`, and `UnicodeMappings`) CPython, IronPython and, in `UnicodeMappings`, Jython and Ruby, obtain better runtime performance than ЯRotor. Therefore, we analyzed the runtime performance of the original SSCLI running these Unicode string tests. We found out that the SSCLI is not very optimized when manipulating Unicode strings: its production version, the CLR, is, on average, 3.6 times faster; and ЯRotor is only 5.4% slower.

IronPython performs better executing some tests (`IfThenElse`, `NestedForLoops`, `ForLoops`, and `DictCreation`) where the CLR is much faster than the original SSCLI, and the reflective features of ЯRotor are not used at all (see Section 6.4 for a more detailed comparison between the SSCLI and the CLR). This fact is also observed in SBCL, which implements a compiler to native code. In 9 of the 30 tests, SBCL has a better runtime performance. These tests do not make extensive use of dynamic features. The exception handling implementation of this language (together with ACL, the other Lisp implementation) involves an important runtime performance penalty. However, if this test is not considered, ЯRotor is still 95.26% faster than SBCL.

	Class-based Model				Prototype-based Model				
Pybench test	**ЯRotor**	**GST**	**ACL**	**SBCL**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
SimpleIntegerArith.	25	11,104	42,033	516	1,344	422	610	1,500	234
SimpleFloatArith.	181	4,799	43,006	31	2,219	485	422	921	266
SimpleIntFloatArith.	135	11,058	44,056	156	1,344	421	609	1,438	250
SimpleLongArith.	59	14,580	22,252	152	1,641	375	469	1,219	281
ConcatUnicode	711	28,968	23,985	4,031	3,797	515	843	1,140	609
CompareUnicode	281	10,792	23,422	15	2,187	422	344	563	312
CreateUnicodeWithC.	446	10,769	13,062	1,172	2,406	250	375	797	343
UnicodeMappings	2,606	6,339	4,672	2,750	1,140	483	516	1,578	2,093
CompareIntegers	15	510	55,606	15	1,953	500	734	1,453	16
CompareFloats	10	531	36,517	15	1,562	453	438	750	16
CompareFloatsInt.	8	343	39,035	15	1,062	468	1,609	766	297
CompareLongs	9	1,274	32,219	15	1,781	1,032	422	1,343	937
CreateNewInstances	40	564	5,562	109	2,016	515	484	1,235	266
NormalClassAttr.	273	1,401	64,048	172	4,593	516	1,016	1,062	1,531
NormalInstanceAttr.	339	1,179	62,985	219	1,234	484	578	969	1,546
SimpleListManip.	81	3,940	31,547	1,406	1,937	422	453	1,359	531
ListSlicing	78	133,443	3,156	63	203	750	781	8,953	1,875
CreateInstances	54	744	6,950	141	2,656	672	657	969	641
TryExcept	27	4,475	1,196,239	1,219,843	922	391	328	219	359
DictCreation	154	18,680	44,187	1,297	4,562	484	500	1,531	109
DictWithStringKeys	52	19,404	43,542	219	3,797	469	469	797	922
DictWithFloatKeys	37	17,052	37,366	266	2,859	1,141	1,250	750	828
DictWithIntegerKeys	39	24,572	45,063	265	1,562	500	531	813	797
SimpleDictManip.	110	3,821	27,156	297	1,609	953	859	1,343	1,062
IfThenElse	15	310	32,278	16	1,937	437	640	2,500	187
NestedForLoops	304	1,020	73,896	78	6,937	594	579	1,062	250
ForLoops	378	632	81,250	78	8,297	453	390	1,078	219
PythonFunctionCalls	226	1,474	34,338	16	375	578	594	1,047	156
PythonMethodCalls	417	895	57,851	141	984	672	656	1,422	1,640
Recursion	66	564	55,656	78	2,390	875	922	1,219	297
Total	**7,200**	**335,237**	**2,368,423**	**1,233,891**	**80,234**	**16,735**	**19,078**	**41,796**	**18,875**

Table 6: Execution time (ms) of the Pybench benchmark.

Class-based Model				Prototype-based Model				
ЯRotor	**GST**	**ACL**	**SBCL**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
10,014,720	12,754,944	78,405,632	290,168,832	26,513,408	8,761,344	9,748,480	53,776,384	58,974,208

Table 7: Memory consumption (KB) of the Pybench benchmark.

6.3.3. The Shootout Benchmark

The last existing benchmark we used to evaluate the selected implementations is the Shootout benchmark (also known as the Computer Language Benchmarks Game) [57]. This benchmark implements different well-known algorithms for statically typed programming languages (no reflection is used at all). We have added dynamic inheritance the same way we did for the Pybench benchmark. We ran the following tests, which do not perform any I/O interaction.

- NBody. Predicts the motion of a group of celestial objects that interact with each other gravitationally.
- Fannkuch redux. This benchmark involves operations (mostly permutations) on vectors of numbers.
- Spectral norm. Calculates the spectral norm (eigenvalue) of a square matrix using the power method.

- Mandelbrot. Calculates a particular instance of the Mandelbrot fractal set.
- Binary trees. Allocates, walks, and deallocates many bottom-up binary trees.

Table 8 gives the details of the performance of the programs selected from the Shootout benchmark, while Table 9 shows the total memory consumption. Similarly, Figure 18 shows the total execution times and memory consumption relative to ЯRotor.

ЯRotor obtained the better runtime performance, but the rest of the languages did not show similar values to before. In this benchmark, where most of the code is statically typed, and dynamic inheritance does not represent an important amount of the total execution time, SBCL was the second fastest implementation after ЯRotor: it was only 40% slower than our system, while it had been over 20 times slower in all of the previous tests. This difference seems to be due to the kind of code executed. Since SBCL implements a high performance compiler [46], its best optimizations are obtained when types are known at compile time. We have also realized that, in those tests where SBCL obtained better runtime performance, the code usually is performing computations using float numbers. This difference is because the JIT-compiler of the SSCLI does not generate optimized code for float number computations. We have compared its performance with that of the CLR, obtaining a difference by a factor of 20, while the average value obtained in Section 6.4 had been 3.34. In order to test this idea, we also developed the Mandelbrot algorithm of the Shootout benchmark (in both ЯRotor and SBCL), using integers instead of floats. The result was that ЯRotor was 3.53 times faster than SBCL (with float numbers, it is the SBCL that is 2.12 times faster than ЯRotor). Finally, SBCL employs 6.55 times more memory than ЯRotor. If we compare the runtime performance ratio to memory consumption, ЯRotor is 4.4 times more efficient than SBCL.

A discussion of the benefits of JIT-compilation can be made by comparing the different implementations of Python. First, we can see how the difference in runtime performance between ЯRotor and CPython increases as the applications use more statically typed code. This difference peaked (3.4 times faster) when running the Shootout benchmark and bottomed out (61.29% slower) when executing our synthetic microbenchmark. Comparing the JIT-compiler IronPython implementation with CPython2, the results are similar. The execution time ratios are: 1.9 (microbenchmark), 1.38 (word count), 1.12 (pybench), and 0.9749 (shootout). This trend was also observed when comparing Jython, which uses the JIT-compiler of the JVM, with CPython 2. Regarding memory consumption, JIT-compiler implementations require more memory to run all the tests, and there is no correlation with the type of benchmark. On average, Jython and IronPython consume 5.4 and 4.7 times more memory than CPython 2.

	Class-based Model				**Prototype-based Model**				
Shootout test	**ЯRotor**	**GST**	**ACL**	**SBCL**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
NBody	344	9,854	2,750	547	3,953	953	889	1,655	625
Fannkuch	172	2,234	446	63	9,625	1,343	1,266	1,985	1,047
SpectralNorm	250	5,126	2,493	203	5,531	1,203	1,905	1,717	1,000
Mandelbrot	47	2,706	584	15	750	171	172	421	109
BinaryTrees	281	1,804	997	360	2,015	531	641	719	234
Total	**1,109**	**21,724**	**7,270**	**1,551**	**21,875**	**4,203**	**4,875**	**6,500**	**3,016**

Table 8: Execution time (ms) of the Shootout benchmark.

Class-based Model				Prototype-based Model				
ЯRotor	**GST**	**ACL**	**SBCL**	**Ruby**	**CPy2**	**CPy3**	**Jy**	**IPy**
6,057,984	10,371,072	47,706,112	45,756,416	6,483,968	5,332,992	8,818,688	38,535,168	34,222,080

Table 9: Memory consumption (KB) of the Shootout benchmark.

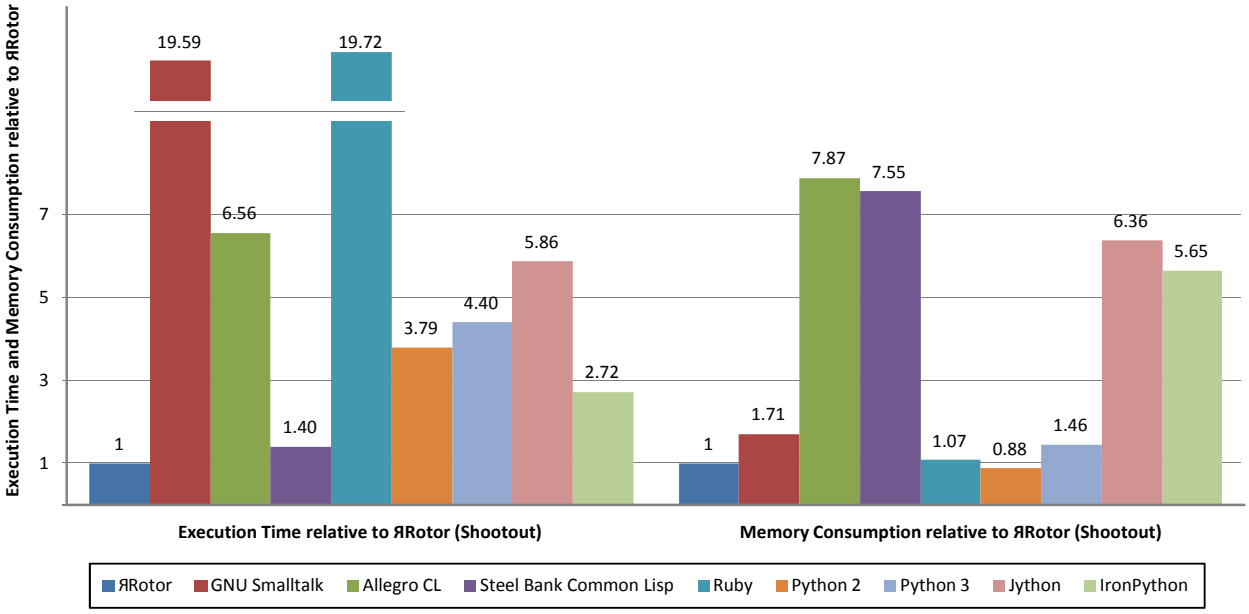

Figure 18: Average execution time and memory consumption relative to ЯRotor, running the Shootout benchmark (lower values are better).

6.4. The Cost of Reflection

This last evaluation section compares the original SSCLI implementation with our platform, using the same programming language (C#). We ran a set of benchmarks that use neither reflection nor dynamic inheritance. The results give us a measurement of the cost of our reflective hybrid class- and prototype-based object-oriented model, when non-reflective class-based statically typed applications are executed. We also compared our base system (the SSCLI) with the production implementation of the CLI: the CLR. This assessment aimed at estimating what might be the efficiency of our system in case it was included in the CLR implementation.

We have selected three existing benchmarks:

- Bruckschlegel. This benchmark was designed by Thomas Bruckschlegel to evaluate the characteristics of Java, C#, and C++ on Windows and Linux. It is composed of a set of 13 elementary tests that use fundamental data processing and arithmetic operations [58].

- Java Grande [59]. A port to C# of a subset of this benchmark suite developed by Chandra Krintz [60]. This port consist of three sections.

    1. Section 1 (low level operations). *Arith*, execution of arithmetic operations; *Assign*, variable, object, and class variables, and array assignments; *Cast*, casting between different primitive types; *Create*, object and array creation; and *Loop*, loop overheads.

2. 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.
3. Section 3 (Large Scale Applications). *RayTracer*, a 3D ray tracer of scenes that contain 64 spheres, and are rendered at a resolution of $N \times N$ pixels, but removing the code that paints these spheres on the screen (only the calculations are performed).

- Zorn. Three real C# applications collected by Ben Zorn [61], consisting of a collection of managed code benchmarks available for performance studies of CLI implementations. These programs are:

  – LCSCBench. Based on the front-end of a C# compiler, it uses a generalized LR (GLR) parsing algorithm. This benchmark is 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 of 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. This program has 10,900 lines of code.

The runtime performances and memory consumption obtained in the execution of these programs are shown in Table 10. The average execution times and memory consumption relative to the SSCLI are displayed in Figures 19 and 20. Both representations show how our implementation involves an average runtime performance cost of 16.45%, and 4.26% more memory utilization than the SSCLI. If we compare the runtime performance of the SSCLI with that of the CLR, the latter is 3.34 times faster than the former, using only 4.23% more memory.

		ЯRotor		SSCLI		CLR	
	Benchmark test	**Time**	**Memory**	**Time**	**Memory**	**Time**	**Memory**
Zorn	LCSCBench	2,130	5,672	2,008	5,375	422	5,320
	AHCBench	1,359	29,729	1,125	26,896	984	34,764
	SharpSATBench	3,328	11,500	2,088	10,358	234	14,942
Java Grande	JGFArithBench	2,188	4,296	2,188	4,036	1,734	3,986
	JGFAssignBench	672	4,316	266	4,308	31	4,212
	JGFCastBench	328	4,564	328	4,244	141	4,212
	JGFCreateBench	6,672	6,498	6,670	6,300	1,609	4,216
	JGFLoopBench	281	4,556	266	4,232	31	4,148
	JGFFFT	13,063	37,247	12,641	36,994	1,813	37,196
	JGFHeapsort	2,406	8,182	1,719	7,924	313	6,158
	JGFSparseMatmult	3,391	8,992	3,313	8,726	438	9,042
	JGFRayTracer	61,438	5,218	38,641	5,016	2,469	4,806
Bruckschlegel	Int arithmetic	640	4,220	640	3,988	296	4,276
	Double arithmetic	2,745	4,640	2,740	4,312	554	4,288
	Long arithmetic	1,750	4,664	1,688	4,336	799	4,288
	Trig	239	4,668	238	4,336	143	5,496
	IO	145	4,668	145	4,336	43	5,496
	Array	203	4,668	200	5,464	14	5,496
	HashMap	17	5,968	17	6,464	5	5,568
	HashMaps	109	5,968	73	6,464	21	5,568
	Heapsort	130	7,328	124	6,464	21	5,568
	Vector	15	7,328	10	6,464	4	5,568
	Matrix Multiply	12,220	8,128	12,220	7,648	1,793	8,988
	Nested Loops	1,047	8,132	1,047	7,648	278	9,000
	String Concat	218	23,952	155	23,468	31	17,936

Table 10: Execution time (ms) and memory consumption (KB) of three different benchmarks that do not use dynamic inheritance.

Although the memory consumption variance is low (6%), the coefficient of variation of the runtime performance penalty is 29.9%. This is because a large number of tests have almost no performance cost, whereas others do. This difference could be clearly seen in the two heap sort algorithm implementations: although the cost is only 4% for the Bruckschlegel benchmark, it is 40% for the Java Grande benchmark. Analyzing the code, we realized that the main difference between them is that the former sorts an array local variable while the latter uses an object field. This result was in contrast to the rest of the tests. Programs where there is a performance penalty are those that perform more accesses to object members. Consequently, we evaluated the member access and method invocation costs with a simple micro-benchmark. The results obtained converged to the following percentages.

- The runtime performance cost of accessing an object's field is 31.65%. This value is 93.12% in the case of `static` (class) fields. The higher performance penalty of `static` field access is due to the worse performance of the search mechanism for this type of member. This penalty is caused by the code we added to provide compatibility with the storage mechanism of the `static` attributes implemented by the SSCLI [18].

- Method invocation involves a performance penalty of 28.29% when the message is sent to an object, and 29.51% when the receiver is a class.

This assessment confirms the performance penalties shown in Figure 19, identifying the places where the addition of dynamic inheritance has involved higher costs. The JGFAssign benchmark

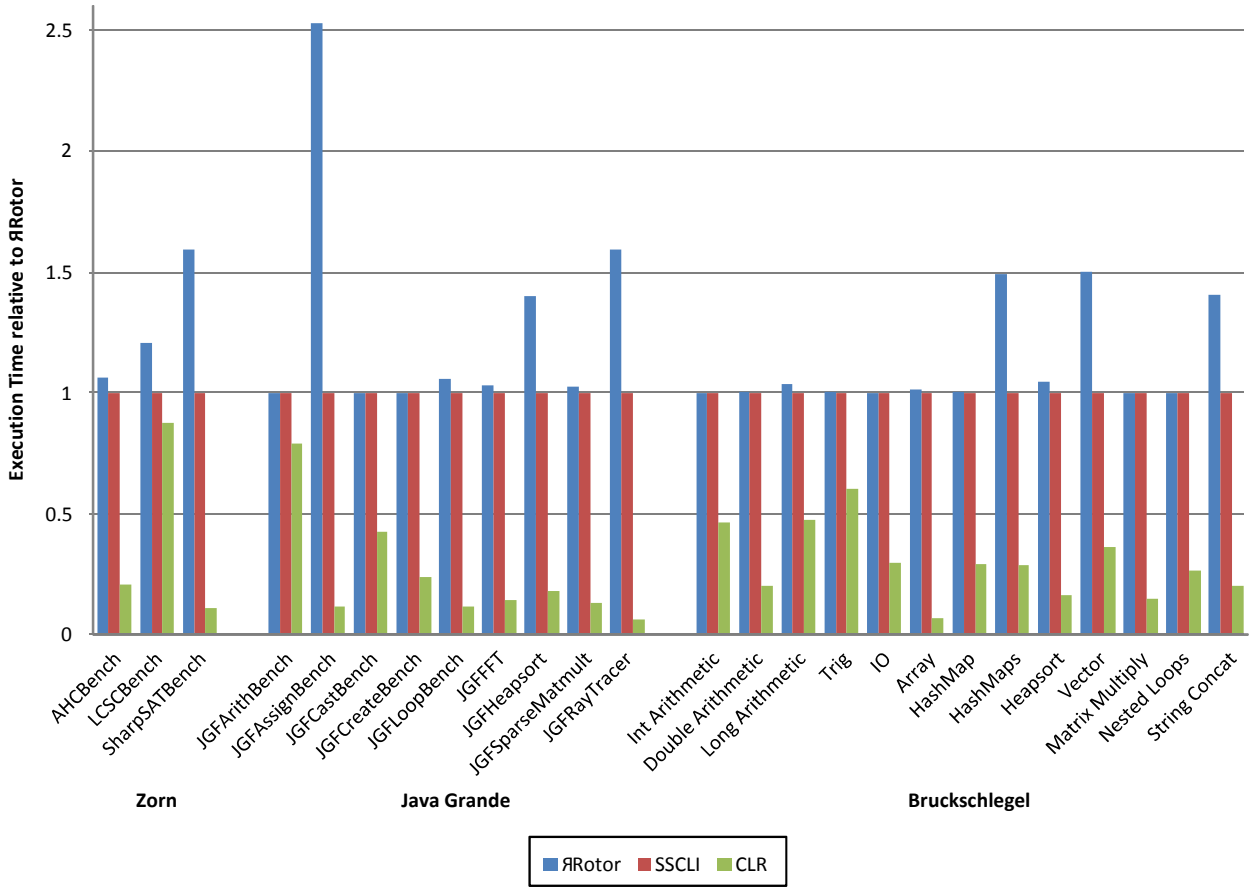

Figure 19: Average execution time relative to SSCLI, running three different benchmarks that do not use dynamic inheritance (lower values are better).

evaluates different kinds of assignments, making a wide use of `static` fields. This is the reason why it has the highest performance penalty (153%). The rest of the tests where the cost is appreciable have penalties lower than 60% (16.45% on average).

7. Related Work

There have been different approaches to optimizing the support of dynamic inheritance. We first analyze those approaches that provide dynamic inheritance using the prototype-based object model, and then the ones that support it in the class-based one—as far as we know, there is no other hybrid system that provides both. Finally, we will analyze those virtual machines aimed at supporting specific features of dynamic languages.

7.1. Prototype-Based Model

Self was the first prototype-based language, in which classes are not present and objects are defined as a collection of slots representing both attributes and methods [33]. It is a reflective language that supports intercession and an advanced JIT-compiler virtual machine that performs

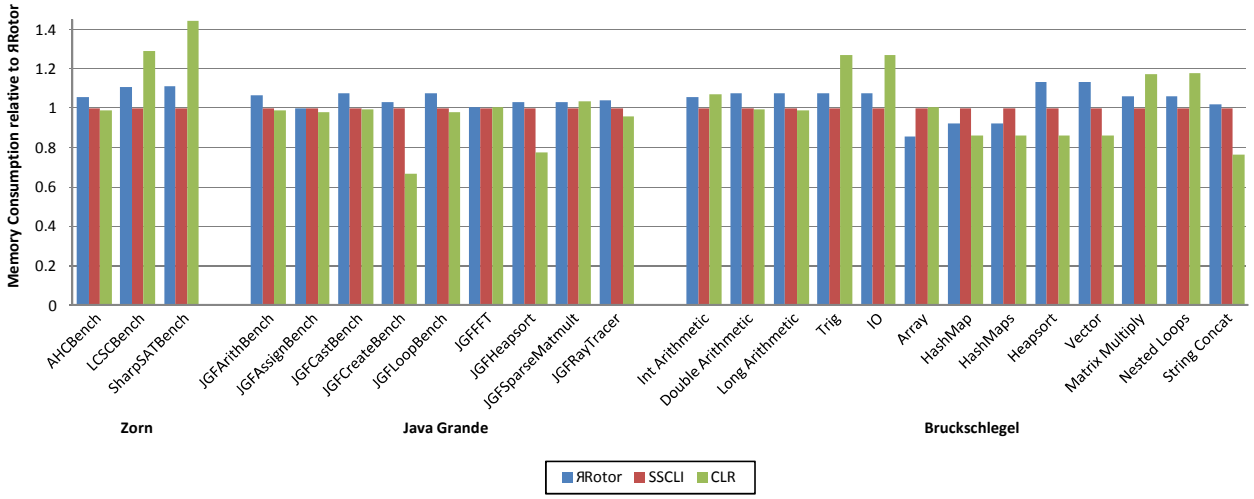

Figure 20: Average memory consumption relative to SSCLI, running three different benchmarks that do not use dynamic inheritance (lower values are better).

at up to half the speed of optimized C [62]. Self provides multiple inheritance. Any slot can be turned into a pointer to a base object (usually called a `parent`) by adding an asterisk as a suffix. Dynamic inheritance is provided by simply changing the value of this pointer. An object delegates any message it does not understand to the parent objects [63], implementing a delegation method-lookup approach.

Cecil is a pure object-oriented programming language developed by Craig Chambers [34]. Cecil implements the prototype-based object model, suppressing the concept of classes—the object is the only abstraction. Cecil incorporates multiple-dispatch methods (multi-methods), and a flexible static type system which allows mixing statically and dynamically typed code. Cecil implements a particular form of dynamic inheritance by means of *predicate objects* [64], a mechanism to capture runtime varying object behavior. When the expression in a predicate object is true, the object associated to that predicate expression is considered as its base object. This controlled change of the inheritance tree at runtime allows performing type checking at compile-time.

Anderson and Drossopoulou [65] defined δ, a simple calculus that provides a formal foundation for an imperative prototype-based system with delegation and dynamic inheritance. They defined 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 Cardelli and Abadi in their ς calculus [20]. In addition to the list of methods, an object contains a collection of addresses pointing to its parents. The operational semantics of δ allows the dynamic addition, update, and modification of this collection, providing a dynamic inheritance mechanism with a delegation-based inheritance strategy.

Python is a mature dynamic language that provides a rich set of reflective features such as structural intercession and dynamic inheritance [35]. Python has many implementations, CPython being the reference one. Python is not commonly referred to as a prototype-based language because it provides the concept of class. However, we have classified it as such because its semantics of structural reflection and dynamic inheritance follows the prototype-based model, as

described in Section 4: an object may hold its own structure, being different from the one defined by its class; besides, adapting the class structure does not involve changing the structure of its instances. Therefore, Python classes are actually traits objects. In Python, the type of an object can be changed through the `__class__` attribute, and base classes can be altered via the `__bases__` one.

7.2. Class-Based Model

CLOS (Common Lisp Object System) is a Lisp extension for object-oriented programming which is part of the ANSI Common Lisp [22]. It is a reflective class-based language that provides intercession and dynamic inheritance. Classes can be redefined by evaluating a new `defclass` form. When a class is redefined, the changes are propagated to its instances and to the instances of any of its subclasses. Class redefinition provides the functionality of changing the inheritance tree by simply declaring a different parent. Instance updating occurs at an implementation-dependent time (eager or lazy), but no later than the next time a slot of that instance is read or written [66]. Instance identity is maintained. The generic function `change-class` changes the class of an instance to a new class. Changing an instance class from *C* to *C'* alters the instance structure to match the new class *C'* structure. This change involves removing all those members of *C* that are not present in the *C'* class. If *C* has any slot with the same name in *C'*, its value is retained. The other new slots are initialized with default values [67].

The Smalltalk approach to implementing class-based dynamic inheritance is quite similar to that of CLOS. The `superclass:` message is used to dynamically modify the class inheritance tree [27]. When a class definition changes, the existing instances are also structurally modified (by the `ClassBuilder` class) in order to match the definition of their new class. Smalltalk also provides the functionality to change the type of an object with the `changeClassTo` method. However, this semantics is not followed in every implementation. Both VisualWorks [47] and Dolphin Smalltalk [48] impose the same restriction on the type change primitive: both the new and old classes must define the same physical structure for their instances [23].

The concept of *wide classes*, defined by Serrano [28], is an extension of the class-based model, which allows instances to be temporarily widened, extending their structure and behavior. The widening operation on a single instance allows it to be temporarily transformed into an instance of a special subclass (a *wide class*). This approach also defines the opposite operation (shrinking an instance), which reshapes the instance to its original class. Widened objects preserve the subtyping relationship, since wide classes are always derived from the original class of the instance. It is possible to widen an object with two disjoint sets of messages and, depending on runtime values, to pass those recently added messages. Since wide classes should be explicitly declared, the type of an instance cannot be widened to an arbitrary subclass. This approach was implemented in the Bigloo programming language, an open implementation of Scheme [68].

$\mathcal{F}$ickle$_{\mathrm{II}}$ is a small class-based language that supports the type change primitive of dynamic inheritance to demonstrate how this feature could be introduced in an imperative, statically typed, class-based, object-oriented language [24]. They define the type change primitive as *dynamic object reclassification*: a programming language feature that allows an object to change its class membership at runtime while retaining its identity. In $\mathcal{F}$ickle$_{\mathrm{II}}$, a class definition may be preceded by the keywords `root` or `state`. Class reclassification can only occur within a hierarchy rooted

with a `root` class. `state` classes are subclasses of `root` classes and they are the only ones that can be reclassified. Classes that are neither `root` nor `state` are respected by reclassification [25]. The $\mathcal{F}$ickle$_{\text{II}}$ implementation of object reclassification offers an advantage over similar approaches (such as wide classes): $\mathcal{F}$ickle$_{\text{II}}$ is type-safe, i.e., any type-correct program (in terms of the type system) is guaranteed never to attempt to access non-existing fields or methods [69].

Predicate classes are a linguistic construct proposed by Chambers [64] aimed at providing transparent type change functionality at runtime based on dynamically evaluated predicates. Like a normal class, a predicate class has a set of superclasses, methods, and attributes. However, unlike a normal class, an object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. That predicate expression can test the state of the object, thus providing a form of implicit property-based dynamic inheritance (type change). This linguistic construct was added to the Cecil programming language [34]. Because Cecil is prototype-based rather than class-based, the adaptation of predicate classes to Cecil's object model was renamed to *predicate objects* (see Section 7.1).

The work implemented by Würthinger *et al.* [70] modifies an early access development build of the Java HotSpot™ VM for Java 1.7 to allow arbitrary changes to the definition of loaded classes. This system follows our approach of modifying a JIT-compiled VM to obtain an efficient implementation of dynamic inheritance. They allow arbitrary changes to the definition of loaded classes, focusing on the class-based object model implemented by the JVM. The static type checking of Java is maintained and dynamic verification of the current state of the program is performed, ensuring the type safety of class hierarchy changes. However, the programmer still has to ensure that the semantics of the modified program is correct, and that the new hierarchy can start running from the state left behind by the previous one [70]. This system does not permit the programmer to change the type of an object to a new class that is loaded at runtime (e.g., using the `Class.forName` method). Dynamic changes are only applied at points where a Java program can be suspended. The instance type change algorithm modifies the garbage collector in order to increase the size of objects, so type change can only be performed at garbage collection time. Runtime performance after code evolution implies an approximate performance penalty of 15%, but the slowdown of the next run after code evolution was measured to be only about 3% [71]. This system is currently the reference implementation of the hot-swapping feature (JSR 292) of the *Da Vinci Machine* project (see Section 7.3).

7.3. Virtual Machines that Support Dynamic Languages

The Dynamic Language Runtime (DLR) [72] 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 [73]. 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, introspection, and object-level intercession by means of a special `ExpandoObject` class. Dynamic inheritance is not provided.

Parrot is an open source virtual machine designed to efficiently compile and execute bytecode for dynamic languages [74]. The virtual machine is register-based rather than stack-based, and employs 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 dynamic inheritance. However, a strong limitation is imposed: these operations throw an exception if the current class has been instantiated [75]. Therefore, neither dynamic inheritance nor intercession allows changing any class structure if the class has any running instance.

The Java Specification Request (JSR) 292 is aimed at supporting dynamically typed languages over the Java platform [15]. 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 C-based 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. 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 [15], it was not finally included. However, the *Da Vinci Machine* (also called the Multi Language Virtual Machine) project [16] 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 the one presented in this paper 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 Zero programming system is a vehicle for learning object-oriented prototype-based programming, and a demonstration of the benefits of container-based persistence [76]. It implements the prototype-based object model, and supports features of dynamic languages such as dynamic inheritance, duck typing, and structural intercession. This system consists of a language-neutral virtual machine (Zero VM), an assembler language, and a standard library. To facilitate programming for the Zero VM, a macro-assembler and the programming languages Prowl and J– are offered. PROWL (PROtotype Writing Language) is a programming language designed to provide a maximum of flexibility during program development: it is prototype-based, dynamically-typed, and structurally intercessive [77]. A distinctive feature of the Prowl language is its mechanism for conditional inheritance, which reduces the programming complexity associated to the use of unrestricted dynamic inheritance. This mechanism was inspired by the predicate objects of the Cecil programming language [34]. The J– programming language is a subset of Java, showing how Zero VM, a prototype-based dynamically typed platform, is able to support class-based statically-typed programming languages such as Java.

The ЯRotor project is aimed at providing direct support of dynamic languages by a JIT-

compiler virtual machine that already provides efficient execution of a wide set of programming languages [78]. First, we developed in C# all the reflective primitives provided by common dynamic languages inside the BCL. Once the primitives were validated in this first prototype, we implemented these primitives in C as part of the execution environment, without altering the BCL interface [36]. Since we used the internal SSCLI structures, data types, and routines, the runtime performance of our previous implementation was greatly improved. The next phase was to modify the JIT compiler in order to extend the semantics of some IL instructions to support structural reflection and duck typing [18]. Accessing the new services in IL instead of calling the BCL library was another significant improvement in its runtime performance. The present paper has presented the next step of this project: adding dynamic inheritance support to the hybrid class- and prototype-based model provided by ЯRotor.

8. Conclusions

The present paper presents a design of dynamic inheritance for both the class- and the prototype-based object models that, implemented in a JIT-compiler virtual machine, was then evaluated as providing significant runtime performance benefits with low memory consumption. The resulting platform supports both dynamically and statically typed languages. Depending on the language to be compiled, compilers can select the appropriate object model, generating code that uses the specific features of dynamic inheritance and structural reflection. Since we have extended the semantics of the base virtual machine, backward compatibility with existing .NET applications was maintained. Instead of implementing an extra layer over the virtual machine that simulates dynamic inheritance and intercession (as does the DLR), we extended its implementation to provide direct support for these features, obtaining significant performance benefits.

In the evaluation of different applications with realistic workloads (written in distinct languages) that made use of dynamic inheritance, our implementation had the highest efficiency. When dynamic inheritance is a notable part of the code executed, it was at least 145% (up to 2,160 times) faster than the other systems evaluated. Running dynamically typed code, the benefit is at least 132% (up to a factor of 328). For static typing, the values obtained range from 40% to 1,972%. The average performance penalty introduced was 16.45%. The measurements carried out suggest that the performance benefits might be increased up to 4.32 times if our proposal were included in the production CLR implementation.

Future work will be focused on retargeting the existing implementation of the *StaDyn* programming language to use ЯRotor as a new back-end [79]. *StaDyn* is a research programming language that supports both dynamic and static typing, extending the semantics of C# [80]. The current implementation generates CLR code, making use of the introspective services offered by the .NET Framework [81]. Future versions including dynamic inheritance and structural intercession will generate both ЯRotor and DLR code. We are also planning to use ЯRotor as a new back-end for the DSAW aspect platform [82, 83]. Structural reflection can be used to obtain flexible dynamic aspect-oriented services [13], and ЯRotor might involve a runtime performance improvement.

The source code of the ЯRotor virtual machine, a binary executable version for Windows, and all the examples and benchmarks used in this paper are freely available at `http://www.reflection.uniovi.es/rrotor/download/2012/jss`

Acknowledgments

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 Zope book, `http://www.zope.org/Documentation/Books/ZopeBook/` (2008).
[8] Django Software Foundation, Django, the web framework for perfectionists with deadlines, `http://openjdk.java.net/projects/mlvm` (2012).
[9] R. Ierusalimschy, L. H. de Figueiredo, W. C. Filho, Lua – an extensible extension language, Software Practice & Experience 26 (1996) 635–652.
[10] R. Ierusalimschy, L. H. de Figueiredo, W. Celes, The evolution of Lua, in: Proceedings of the conference on History of Programming Languages (HOPL), San Diego, California, 2007, pp. 1–26.
[11] J. Hermann, The Pythius Web Site, `http://pythius.sourceforge.net` (2012).
[12] K. Böllert, On weaving aspects, in: Proceedings of the Workshop on Object-Oriented Technology, 1999, pp. 301–302.
[13] F. Ortin, J. M. Cueva, Dynamic adaptation of application aspects, Journal of Systems and Software (2004) 229–243.
[14] J. Hugunin, Just glue it! Ruby and the DLR in Silverlight, in: MIX'2007, 2007.
[15] Oracle, JSR 292, supporting dynamically typed languages on the Java platform, `http://www.jcp.org/en/jsr/detail?id=292` (2011).
[16] Oracle, The Da Vinci Machine, a multi-language renaissance for the Java virtual machine architecture, `http://openjdk.java.net/projects/mlvm` (2012).
[17] B. C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, Massachusetts, 2002.
[18] 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.
[19] C. Lucas, K. Mens, P. Steyaert, Typing dynamic inheritance: A trade-off between substitutability and extensibility, Tech. Rep. vub-prog-tr-95-03, Vrije Universiteit Brussel (1995).
[20] M. Abadi, L. Cardelli, A Theory of Objects, Springer, New York, 1998.
[21] A. Taivalsaari, On the notion of inheritance, ACM Computing Surveys 28 (1996) 438–479.
[22] L. G. DeMichiel, R. P. Gabriel, The Common Lisp object system: An overview, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1987, pp. 151–170.
[23] F. Rivard, Smalltalk: a reflective language, in: Proceedings of Reflection 96, 1996, pp. 21–38.

[24] 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.
[25] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca, A type preserving translation of Flickle into Java, in: Electronic Notes in Theoretical Computer Science, Vol. 62, 2002, pp. 69–82.
[26] 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.
[27] A. Goldberg, D. Robson, Smalltalk-80: the language and its implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.
[28] M. Serrano, Wide classes, European Conference on Object-Oriented Programming, Lecture Notes in Computer Science 1628.
[29] A. H. Borning, Classes versus prototypes in object-oriented languages, in: Proceedings of the ACM/IEEE Fall Joint Computer Conference, 1986, pp. 36–40.
[30] 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.
[31] M. Wolczko, O. Agesen, D. Ungar, Towards a universal implementation substrate for object-oriented languages, Oracle Laboratories (`http://labs.oracle.com/people/mario/pubs/substrate.pdf`) (1996).
[32] H. Lieberman, Using prototypical objects to implement shared behavior in object-oriented systems, in: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), OOPLSA, ACM, New York, NY, USA, 1986, pp. 214–223.
[33] 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.
[34] C. Chambers, Object-oriented multi-methods in Cecil, in: European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, 1992, pp. 33–56.
[35] Python, The official Python website, `www.python.org` (2012).
[36] J. M. Redondo, F. Ortin, J. M. Cueva, Optimizing reflective primitives of dynamic languages, in: International Journal of Software Engineering and Knowledge Engineering, Vol. 18, World Scientific, 2008, pp. 759–783.
[37] D. Stutz, T. Neward, G. Shilling, Shared Source CLI Essentials, O'Reilly, 2003.
[38] ECMA, ECMA-335 standard: Common Language Infrastructure, `http://www.ecma-international.org/publications/standards/Ecma-335.htm` (2009).
[39] ECMA, ECMA-334 standard: C# language specification 4th edition, `http://www.ecma-international.org/publications/standards/Ecma-334.htm` (2009).
[40] A. Taivalsaari, Delegation versus concatenation or cloning is inheritance too, ACM SIGPLAN OOPS Messenger 6 (1994) 20–49.
[41] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, 1995.
[42] PythonSF, Jython: Python for the Java platform, `http://www.jython.org` (2012).
[43] Mono-Project, The Mono project, `http://www.mono-project.com` (2012).
[44] Prodata, The benchmarks of some Common LISP implementations, `http://www.prodata.lt/EN/Programming/lisp_bmarks.html` (2012).
[45] Franz, Allegro Common Lisp 8.1 official website, `www.franz.com/products/allegrocl` (2009).
[46] SBCL, Steel Bank Common Lisp homepage, `http://www.sbcl.org` (2012).
[47] Cincom, Visualworks Smalltalk homepage, `http://www.cincomsmalltalk.com/main/products/visualworks/`) (2012).
[48] ObjectArts, Dolphin Smalltalk official homepage, `http://www.object-arts.com/` (2012).
[49] D. Berger, Ruby code that will swallow your soul!, O'Reilly Ruby.
[50] Y. Matsumoto, Ruby programming language homepage, `http://www.ruby-lang.org/en/` (2011).
[51] F. Gross, Evil Ruby project home page, `http://rubyforge.org/projects/evil/` (2012).
[52] MicrosoftTechnet, Windows server techcenter: Windows performance monitor, `http://technet.microsoft.com/en-us/library/cc749249.aspx` (2012).
[53] 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.
[54] C. Chambers, The design and implementation of the Self compiler, an optimizing compiler for object-oriented programming languages, Ph.D. thesis, Stanford University (1992).
[55] O. Agesen, J. Palsberg, M. Schwartzbach, Type inference of Self: Analysis of objects with dynamic and multiple inheritance, Software-Practice and Experience 25 (1995) 975–995.
[56] P. S. Foundation, Pybench benchmark project trunk page, http://svn.python.org/projects/python/trunk/Tools/pybench/) (2012).
[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] 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.
[60] C. A. Krintz, A collection of phoenix-compatible C# benchmarks, http://www.cs.ucsb.edu/~ckrintz/racelab/PhxCSBenchmarks (2012).
[61] B. Zorn, CLI benchmarks, Microsoft Research (http://research.microsoft.com/en-us/um/people/zorn/benchmarks) (2012).
[62] C. Chambers, D. Ungar, Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language, in: Conference on Programming language design and implementation (PLDI), 1989, pp. 146–160.
[63] C. Chambers, D. Ungar, B.-W. Chang, U. Hï£¡lzle, Parents are shared parts of objects: Inheritance and encapsulation in Self, in: Lisp and Symbolic Computation, 1991, pp. 207–222.
[64] C. Chambers, Predicate classes, in: European Conference on Object-Oriented Programming (ECOOP), 1993, pp. 268–296.
[65] C. Anderson, S. Drossopoulou, δ: an imperative object based calculus, in: International Workshop on Unanticipated Software Evolution (USE), 2002.
URL http://pubs.doc.ic.ac.uk/deltacalc/
[66] F. P. Miller, A. F. Vandome, J. McBrewster, Common Lisp: Lisp (programming language), Programming language, American National Standards Institute, Specification (technical standard), Free and open source software, Programming paradigm, Alpha Press, 2010.
[67] N. Levine, Fundamentals of CLOS, Ravenbrook Limited (http://cl-cookbook.sourceforge.net/clos-tutorial/index.html) (2003).
[68] M. Serrano, Bigloo. A practical Scheme compiler. User manual for version 3.8a., http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.pdf (2012).
[69] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, E. Zucca, A provenly correct translation of Fickle into Java, ACM Transactions on Programming Languages and Systems 29. doi:http://doi.acm.org/10.1145/1216374.1216381.
[70] T. Würthinger, C. Wimmerb, L. Stadler, Unrestricted and safe dynamic code evolution for Java, Science of Computer Programming.
[71] T. Würthinger, C. Wimmer, L. Stadler, Dynamic code evolution for Java, in: Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java, PPPJ '10, ACM, New York, NY, USA, 2010, pp. 10–19.
[72] B. Chiles, Common Language Runtime inside out: IronPython and the Dynamic Language Runtime, msdn2.microsoft.com/en-us/magazine/cc163344.aspx (2008).
[73] G. M. Bierman, E. Meijer, M. Torgersen, Adding dynamic types to C#, in: European Conference on Object-Oriented Programming (ECOOP), 2010, pp. 76–100.
[74] Parrot, Parrot VM homepage, http://www.parrot.org (2012).
[75] Parrot, Class PMC implementation, http://docs.parrot.org/parrot/devel/html/src/pmc/class.pmc.html (2012).
[76] J. B. García Perez-Schofield, F. Ortin, E. García Roselló, M. Pérez Cota, Towards an object-oriented programming system for education, Computer Applications in Engineering Education 14 (4) (2006) 243–332.

[77] 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.
[78] F. Ortin, J. M. Redondo, L. Vinuesa, J. M. Cueva, Adding structural reflection to the SSCLI, in: Conference on .Net Technologies, 2005, pp. 151–162.
[79] 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.
[80] F. Ortin, M. García, Union and intersection types to support both dynamic and static typing, Information Processing Letters 111 (6) (2011) 278–286.
[81] F. Ortin, Type inference to optimize a hybrid statically and dynamically typed language, The Computer Journal 54 (11) (2011) 1901–1924.
[82] 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.
[83] L. Vinuesa, F. Ortin, J. M. Felix, F. Alvarez, DSAW: A dynamic and static aspect weaving platform, in: Proceedings of the International Conference on Software and Data Technologies (ICSOFT), INSTICC, 2008, pp. 55–62.