A dynamic virtual machine for the support of interoperable programming languages

J. Baltasar García Perez-Schofield
Faculty of Computer Science, University of Vigo
Edificio Politécnico, s/n, Campus As Lagoas
32004 Orense(Spain)
+34 988 36 88 91
jbgarcia@uvigo.es

Francisco Ortín Soler
Technical School of Computer Science, Oviedo
Department of Computer Science, Calvo Sotelo s/n.
33007. Oviedo (Spain)
+34 985 10 31 72
ortin@lsi.uniovi.es

ABSTRACT
In this paper, the Zero project, a persistent, prototype-based programming system, is discussed. Zero has been in continuous development since 2003, and in use in advanced subjects at the University of Vigo, Computer Science faculty. Its main characteristic is that it has been designed with multi-language support in mind from the beginning. Its approach is based in a dynamic, flexible virtual machine with native persistence capabilities, which can support programming languages of different natures in regard to their type systems: static, dynamic or even hybrid.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications - extensible languages, object-oriented languages.

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Algorithms, Performance, Design, Languages, Persistence.

Keywords
Dynamic programming languages, static programming languages, type systems.

1. INTRODUCTION
In this paper, we the authors use the terms *static* and *dynamic* very often. The context in which these terms are used is important, since both of them are used in many different areas of research. The use of static in regard to programming languages refers here to those languages that have a strong type verification at compile time. Some of these programming languages even support introspection. For example, C++ supports a very limited introspection mechanism called RTTI (Run-Time Type Identification) [8], while Java [1] or C# [3] fully support it. On the other hand, dynamic programming languages typically do not carry out any type checking at compile time. In many cases (such as *Self* [9], or *Prowl* [5] itself, discussed later), there are even no classes nor types. The most widely-known reference is currently Python [6].

The main advantage of static programming languages is that performing type checking at compile time, reveals many errors that could have been originated even in a simple identifier spelling. The main advantage of dynamic programming languages is what is known as structural reflection: the program itself is modifiable at run-time. Thus, even some kind of maintenance tasks can be carried out without having to stop the application.

In very recent years, various attempts to close the gap between static and dynamic languages have been performed. One widely-known reference is probably *IronPython* [4]*:* it is an implementation of a dynamic programming language in a statically-typed virtual machine such as the .NET CLR (Common Language Runtime). However, this programming language has the undesirable characteristic of being unable to share the created classes and objects with other programming languages supported by the .NET CLR, such as C# or Visual Basic. This is due to its impossibility of using the CTS (Common Type System) of the .NET framework, because of the lack of support for structural reflection within the .NET CLR. Classes must be represented by special structures that resemble the ones that the Python virtual machine (CPython) must have to maintain in memory during program execution, which obviously have nothing to do with native CLR classes.

Other attempts, even in classical languages such as C++, or more modern ones such as C#, include a very limited kind of type inference (the recycled keyword *auto* in C++, or *var* in C#). This means that, in the specific context of declaration of references, the type of the reference does not have to be given, when this possibility is used it is taken from the type of the object (the r-value) at compile time.

1.1 Background
Our main projects of research are Reflective Rotor [7] and Zero [5] (the one in which this paper is centered). The first one is a severe modification of the SSCLI (the shared source version of the .NET CLR) in order to make it accept structural reflection. Although this project was concluded successfully, we learned about the difficulties related to the modification of an existing VM (virtual machine): a) it is difficult to express something that is not native, leading to the need of very tricky hacks, which b) are extremely complex to maintain, and c) they are difficult to simply update them for new versions of the VM.

Zero is a programming system which does not provide types nor classes. There are prototypical objects which can be assumed to be classes (more or less the kind of distinction *Python* does), and from which derive the instances themselves. For example, all text strings derive from the object *String*, which derives from the

object *DataType*, which itself derives from *Object*, the root of the hierarchy. All method calls are implemented as of the late-binding kind, since, due to its support of structural reflection, the message can be sent to a method that does not exist at compile time, but that will exist later at run-time.

Zero supports two programming languages: *Prowl*, which is the most native one, and *J--*, a subset of the Java programming language, with strong type verification.

1.2 Motivation

As discussed before, both approaches have their advantages and disadvantages: the main disadvantage of static languages comes from its lack of flexibility, while the main disadvantage of dynamic languages comes from its lack of any kind of type checking at compile time. However, it is known that dynamic languages such as *Self*, *Prowl*, etc., can model the possibilities of the static ones, while the reverse is not true. Their design uses prototypes instead of classes, which have more express power [10]. The motivation of the authors is to effectively provide a common substrate for different kinds of programming languages, and as future work, design a type system that can be added on demand to the common back-end for all of them, *za*.

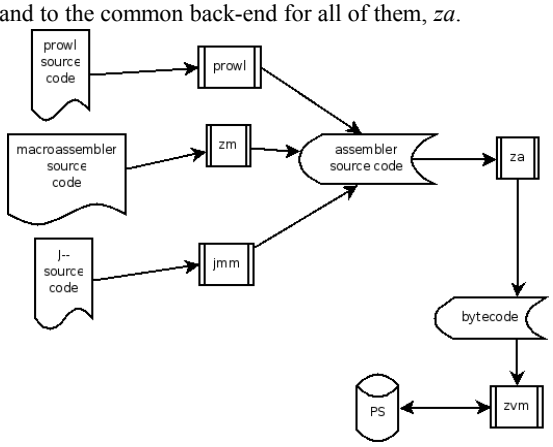

Figure 1: Schematics summarizing the Zero programming system.

The rest of this position paper is structured as follows: firstly the zero programming system is discussed, specially in its features of multi-language support, and finally the conclusions and future work are shown.

2. The Zero Programming System

The core of the system is *za* and *zvm*. The first one is the back-end for the available compilers in the system, and it understands Zero assembler as text. The assembler converts the mnemotechnic words in assemblies (as shown in Figure 1), while also carrying out all optimization techniques possible. Given that Zero is a register-based VM, the main optimization is to reduce the use of local references in methods, taking advantage of the four general-purpose registers available (*__gp1* to *__gp4*), as well as the accumulator (*__acc*) in which the reference to the object result of the last message is stored. The *zvm* unit is the virtual machine. It reads assemblies, checks their integrity, and executes them. All objects are, or can be, persistent (there are no files for data storage nor any other purpose), so it takes care of the persistent store as well.

The available programming languages are *Prowl* and *J--* (there is even a macroassembler, *zm*, allowing a higher level of abstraction for programming than the one *za* offers). All of them produce Zero assembler, which is consumed by *za*. *Prowl* is not problematic since it just represents the semantics of the VM (prototype-based, dynamic and no type checking at compile time), while J-- is the strong-typed one (static, class-based and with type checking at compile time). Zero[1] is not limited to only these two programming languages, but both are the ones implemented at the time of writing this paper, being able to share the same computational model. Their abstractions can be translated to the needs of the Zero VM and executed transparently.

The main difficulty is to be able to extract types in a type-agnostic ecosystem (in this section, the word *type* is abused, as the *J--* programming language needs types, while the VM does not know anything about that concept). The *J--* programming language can compile its own programs without difficulties by means of applying regular type checking. However, it still persists the need of knowing types in other assemblies. This need is actually central for the compiler: it involves the standard library, for instance. The compiler needs to load the "external" assembly and analyze it using introspection, and only then type-checking can be carried out at compile time.

```
class Point extends Object {
    public int x;
    public int y;

    //void setX(String xx) // compile-time error
    void setX(int xx)
    {
        x = xx;
    }
    void setY(int yy)
    {
        y = yy;
    }
    String toString()
    {
        String toret = x.toString();
        toret += ", ";
        toret += y.toString();
        return toret;
    }
}

class ChkInteropWrite extends ConsoleApplication
{
    public static void doIt()
    {
        Point p = new Point();
        PersistentStore ps =
                System.getPersistentStore()
        ;
        Container root = ps.getRoot();

        p.setX( 100 ); // compile-time error
        p.setY( 200 );
        root.addRenamedObject( p, "p1" );
    }
}
```

Figure 2: A class Point declared in the J-- programming language.

The interesting point about both programming languages is that the data stored by means of *Prowl* in the persistent store is transparently accessible from *J--* and *viceversa*, as shown in

[1] Available on-line at http://webs.uvigo.es/jbgarcia/prjs/zero

Figures 2 and 3. Firstly, a *J--* program creates a class *Point,* as well as an object *Point* (referenced by the local reference *p*), which is finally stored as *p1* in the main container [5] of the persistent store. Finally, a second program, written in *Prowl*, uses the *Point* class (which has been translated into a prototype) in order to create a new object by means of copying the *Line* prototype, using the point stored as origin, and a new one as end. Finally, the information about the *line* object is shown on the standard console.

```
object Line
        attribute + org = PSRoot.Point;
        attribute + end = PSRoot.Point;

        method + setOrg(o)
        {
                org = o;
                return;
        }
        method + setEnd(e)
        {
                end = e;
                return;
        }
        method + toString()
        {
                reference toret = org.toString();
                toret += " to ";
                toret += end.toString();
                return toret;
        }
endObject

object ChkInteropRead
        method + doIt()
        {
                reference p2 =
                        PSRoot.Point.copy( "" )
                ;
                p2.setX( 110 );
                p2.setY( 220 );

                reference line = Line.copy( "" );
                line.setOrg( PSRoot.p1 );
                line.setEnd( p2 );

                System.console.writeLn( line );
                return;
        }
endObject
```

Figure 3: A small program in Prowl uses the Point class stored before in the PS with J--

The only disadvantage for this process is the extra compilation time needed to carry out the introspective analysis of other assemblies. The compiler could store in a cache already inspected assemblies (such as the so common standard library), and analyze them just in case of modification; however, this compilation overhead would still persist for assemblies not used before.

3. Conclusions and future work

In this paper, the Zero programming system and the already available programming languages, *Prowl* and *J--* have been presented. Though both languages are absolutely different (they could be thought as the extremes of the dynamic/static spectrum), they can a) share the same ecosystem in which types are actually not recognized; they also can b) share data of any kind; objects can be read and written by both languages; due to c) a transparent, language-independent persistence system. The persistence system of Zero, as a transparent representation of objects, is mandatory in order to achieve the objectives shown here.

As future work, it would actually be very interesting to be able to apply that type checking technique not only for programs written in *J--*, but also to programs written in *Prowl*. In other words, move the introspective analysis to the the common back-end: *za*, using type inference at assembly level. For example, it could be activated by a command switch, and therefore be used or not on demand, exposing, for example, spelling errors that obviously do not have their origin in any use of the flexibility provided. This would actually mean the use of a common pluggable type system [2]. The challenge of this proposal would be to provide understandable error messages when necessary, as the back-end, as discussed before, works at assembly level.

4. REFERENCES

[1] Arnold, K.; Gosling, J. *The Java(TM) Programming Language*. Prentice Hall; 4th edition (August 27, 2005). ISBN 978-0321349804.

[2] Bracha G. Pluggable Type Systems. *OOPSLA04 Workshop on Revival of Dynamic Languages.* October 17, 2004.

[3] Heljsberg, A.; Wiltamuth, S.; Golde, P. *The C# Programming Language*. Addison-Wesley Professional (October 30, 2003). ISBN 978-0321154910

[4] Hugunin, J. IronPython: A fast Python implementation for .NET and Mono. *Proceedings of PyCon*. Python Software Foundation, March, 2004

[5] García, B.; Ortín, F.; Roselló, E.; Pérez, C. Visual Zero: A persistent and interactive object-oriented programming environment. *Journal of Visual Languages & Computing*, n19, pp 280-398, 2008

[6] Lutz, M.; *Learning Python: Powerful Object-Oriented Programming*. O'Reilly Media; 4th edition (September 24, 2009). ISBN 978-0596158064

[7] Ortin. F.; Redondo, J.M.; García, B. Efficient virtual machine support of runtime structural reflection. *Science of Computer Programming*, Volume 74, Issue 10, pp. 836-860. Elsevier, August 2009. DOI = http://doi.acm.org/10.1016/j.scico.2009.04.001

[8] Stroustrup, B.; *The Design and Evolution of C++*. Addison-Wesley Professional (April 8, 1994). ISBN 978-0201543308

[9] Ungar, D.; Smith, R. B. Self: The Power of Simplicity. *ACM SIGPLAN Notices* 22(12), December 1987. Notices. DOI = http://doi.acm.org/10.1145/38807.38828

[10] Wolcko, M., Agesen, O., and Ungar, D. 1996. *Towards a Universal Implementation Substrate for Object-Oriented* Languages. Sun Microsystems Laboratories.