Optimización de las Primitivas de Reflexión Ofrecidas por los Lenguajes Dinámicos

José Manuel Redondo López [1,2]

Dpto. de Informática
E. Universitaria de Ingeniería Informática de Oviedo (EUITIO), Univ. de Oviedo
Oviedo, España

Francisco Ortín Soler, Juan Manuel Cueva Lovelle [3]

Dpto. de Informática
E. Universitaria de Ingeniería Informática de Oviedo (EUITIO), Univ. de Oviedo
Oviedo, España

Abstract

Los lenguajes dinámicos permiten realizar un extenso elenco de modificaciones sobre la estructura y comportamiento de un programa y por ello son un medio adecuado de desarrollo para aquellas aplicaciones que necesitan de un alto grado de flexibilidad. Sin embargo, el menor rendimiento de los programas escritos en estos lenguajes puede suponer que no sean elegidos para determinados desarrollos. Exploraremos en este trabajo cómo pueden emplearse máquinas virtuales dotadas de compilación *JIT* para incrementar el rendimiento de las aplicaciones desarrolladas con lenguajes dinámicos, implementando primitivas de reflexión sobre ellas y analizando los resultados obtenidos.

Key words: Reflexión, máquina virtual, rendimiento, l. dinámicos

1 Introducción

Los lenguajes dinámicos (como *Python* [20]) son aquellos que permiten examinar y efectuar modificaciones sobre su propia estructura, comportamiento

[1] Este proyecto ha sido parcialmente financiado por *Microsoft Research* (*Extending Rotor with Structural Reflection to support Reflective Languages*, (04-05)), el Ministerio de Ciencia y Tecn. (Programa Nacional para la Investigación, Desarrollo e Innovación, TIN2004-03453) y por la Univ. de Oviedo (UNOV-06-MB-534).
[2] Email:redondojose@uniovi.es
[3] Email:ortin@lsi.uniovi.es, cueva@uniovi.es

This paper was generated using LATEX macros provided by
Electronic Notes in Theoretical Computer Science

y entorno, con características que le permiten su automodificación y la generación dinámica de código. Hoy en día es posible encontrar distribuciones muy completas de los mismos, válidas para desarrollar muchos tipos de aplicaciones diferentes. Estos lenguajes permiten a las aplicaciones un alto grado de flexibilidad, y podrán por tanto responder mejor ante la aparición de nuevos requisitos o reglas de negocio, algo que sería mas costoso de obtener mediante el empleo de lenguajes "estáticos" tradicionales como *C++*. Existen múltiples ejemplos de aplicaciones desarrolladas con lenguajes dinámicos, orientadas a cubrir necesidades diversas, como *Zope* [21], *software* empleado para la construcción de aplicaciones personalizadas y sistemas de gestión de contenidos, y pudiéndose mencionar también los lenguajes para el desarrollo *Web* como otro ejemplo válido. No obstante, estos lenguajes no están tan extendidos como los "estáticos" para el desarrollo de aplicaciones, aunque su uso pueda reportar ventajas por las características de la propia aplicación a desarrollar o de su entorno. El principal motivo es la carencia de rendimiento en ejecución de las aplicaciones finales, debido a la necesidad de realizar operaciones adicionales, como inferencia de tipos, que no son necesarias en un lenguaje "estático". Esto es un factor que puede tener un gran peso en el desarrollo y que haga que otras consideraciones, como la productividad [4], permanezcan en un segundo plano a la hora de elegir un lenguaje [4].

Por otra parte, cabe destacar cómo las máquinas virtuales han experimentado un constante avance en términos de rendimiento, gracias a técnicas como la compilación bajo demanda (*JIT*) con optimización adaptativa (*HotSpot* [14]), que han permitido aumentar su aplicación práctica. Este trabajo pretende aunar ambos factores, y comprobar si es posible integrar en una máquina virtual "estática" [5] de este tipo un soporte adecuado para primitivas reflectivas de lenguajes dinámicos, intentando así lograr una mejora en sus tiempos de ejecución y comparando los resultados con los que un lenguaje dinámico eficiente proporcione ante idénticas pruebas.

El artículo se estructura como sigue: Se hará una descripción de la máquina virtual empleada, para posteriormente hablar de los lenguajes dinámicos y los problemas que las modificaciónes hechas a la máquina para soportarlos pueden introducir. Finalmente se describirá la implementación realizada y presentaremos los resultados obtenidos, el trabajo futuro y las conclusiones.

2 SSCLI

A la hora de escoger una máquina virtual a modificar, se necesitará un sistema conocido, a ser posible de uso profesional, con facilidades de modificación (tanto a nivel de estructura como de documentación, y también con una li-

[4] En el sentido de que un lenguaje dinámico podría disminuir el tiempo de desarrollo de una aplicación gracias a sus características.

[5] Aquella que es diseñada, implementada y optimizada teniendo sólo en cuenta a los lenguajes estáticos, maximizando su rendimiento en ejecución.

cencia que lo permita), robusto, estable y que cuente con una versión que no esté sometida a constantes actualizaciones. De entre todos los posibles sistemas que pueden encajar con estos requisitos, finalmente se consideró que una plataforma basada en el estándar *CLI* sería lo más adecuado frente a otras opciones como *JVM* (*Java Virtual Machine*), dado que integra características como la generación de código en tiempo de ejecución, la interoperabilidad entre los diferentes lenguajes soportados y la existencia de versiones que permiten la modificación sin restricciones del código de la máquina virtual, que hacen que se ajuste mejor al trabajo que se va a llevar a cabo. De todas las posibles implementaciones de este estándar candidatas se ha seleccionado *SSCLI* (*Shared Source Common Language Infrastructure*, también conocido como *Rotor*, de *Microsoft*) [13], debido a su gran parecido con la distribución comercial *CLR* (compartiendo gran parte de su arquitectura), su elevado rendimiento, sus posibilidades de modificación (incluyendo una licencia *Shared Source* que no establece restricciones significativas al respecto) y por la existencia de una versión estable, no sometida a cambios y evoluciones constantes, sobre la que poder hacer modificaciones.

3 Lenguajes dinámicos y su integración en la máquina

A la hora de integrar un soporte para lenguajes dinámicos dentro de la máquina seleccionada, se han fijado una serie de características de los mismos que deben estar representadas en el sistema modificado [11]:

- `Sistemas de tipos dinámicos`: En aras a mantener su flexibilidad en tiempo de ejecución, los lenguajes dinámicos, una vez compilados, hacen comprobación e inferencia de tipos en tiempo de ejecución, permitiéndose pues que un identificador modifique su tipo en función del camino seguido por la ejecución del programa. Esto puede ocasionar la detección de errores de programación en tiempo de ejecución, que pueden disminuirse empleando programas de test automatizados que ayuden a la detección de errores, como *Junit* [10] o analizadores estáticos de programas.
- `Capacidad para examinar y cambiar su estructura y/o comportamiento`: Esta característica es precisamente la que dota a los lenguajes dinámicos de un alto grado de flexibilidad. Mediante su uso, un programa podría cambiar el conjunto de métodos y/o atributos de cualquiera de sus tipos o instancias o la semántica de un conjunto de primitivas del sistema, reflejándose el resultado de los cambios en la propia ejecución del programa inmediatamente. La reflexión es el mecanismo utilizado para hacer este tipo de operaciones, y el nivel de la misma implementado por el lenguaje [11] determinará que tipo de operaciones se podrán realizar. Gracias a ella, un lenguaje dinámico es un medio más adecuado para crear *software* adaptativo y adaptable [6].
- `Integración de diseño y ejecución`: En un lenguaje verdaderamente

dinámico la barrera existente entre tiempo de diseño y tiempo de ejecución es muy delgada. Si un programa puede efectuar cambios a su estructura o a su semántica, y ésos cambios se reflejan inmediatamente en el propio programa, cualquier cambio realizado mientras el programa esté en funcionamiento tendrá un impacto directo en la propia ejecución del mismo, por lo que ya no se puede establecer una diferenciación clara.

- `Manipulación de diferentes entidades como objetos de primera clase`: Los lenguajes dinámicos convierten muchas de sus entidades (módulos, tipos, clases, objetos,...) así como los elementos que las forman (métodos y atributos) en objetos de primera clase. Esto hace que el código pueda factorizarse de un modo superior respecto a lenguajes que no ofrezcan estas características.

Por tanto la tarea más compleja planteada en este trabajo es transformar el modelo computacional de la máquina virtual, convirtiéndolo en otro que pueda soportar coherentemente el elenco de lenguajes "estáticos" ya existentes conjuntamente con una serie de lenguajes dinámicos, que podrían ejecutarse de forma nativa una vez sea modificada. Además debe tenerse en cuenta que las aplicaciones en diferentes lenguajes deben poder interactuar entre sí, al tener *SSCLI* la característica de interoperabilidad, que permite emplear cualquier entidad desarrollada en un lenguaje X desde otro lenguaje Y (pudiendo ser ahora cualquiera de ellos dinámico). Sin embargo, el modelo de clases utilizado por el sistema no se adapta perfectamente a las operaciones reflectivas planteadas. Esto es así ya que en este modelo se parte de la premisa de que las clases deben representar exactamente la estructura y comportamiento de todas sus instancias. Por tanto, si una clase es modificada, todas sus instancias deberán ser actualizadas consecuentemente, manteniendo así la coherencia del modelo computacional. Este proceso de actualización ha sido identificado en el campo de las bases de datos como evolución de esquemas, y puede hacerse en el momento en que la clase es modificada (modelo ansioso) o bien cuando una clase va a usar dicha modificación (modelo perezoso) [17].

No obstante existe otro problema, cuya solución es más compleja, y que se origina cuando la modificación se produce en una instancia concreta. En este caso el objetivo es que sólo dicha instancia se vea modificada, sin afectar por tanto ni a su clase ni al resto sus instancias "hermanas". Esto ocasionaría que la clase ya no representase la estructura de una de sus instancias, y que por tanto se pierda la coherencia del modelo computacional. Este mismo problema se trató de solucionar en el desarrollo de la plataforma reflectiva *Metaxa* [5], empleando un concepto también proveniente del campo de las bases de datos, denominado versionado de esquemas [12]. Esta solución consiste básicamente en generar una clase "sombra" a partir de la original, qué sea el nuevo tipo de cada instancia modificada. Aunque de esta forma se mantendría la coherencia del modelo computacional, la solución no es operativa dada la elevada complejidad que introduce, ya que es muy sencillo tener que generar un gran numero de clases sombra que pueden introducir un impacto negativo en el rendimiento

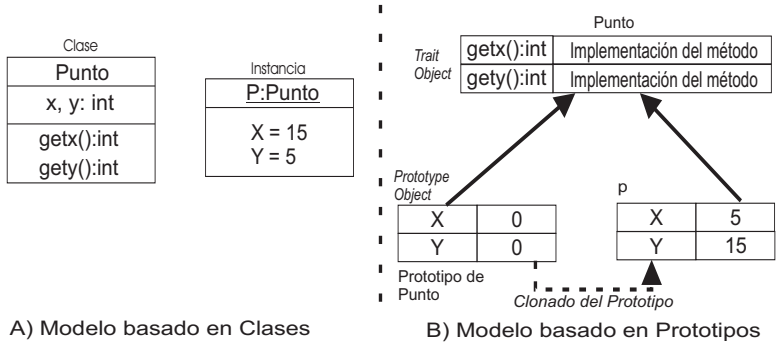

Fig. 1. Estructuración de entidades en el modelo de clases y el de prototipos

y complicaciones a la hora de manipular tipos.

Por tanto, se ha tenido que buscar una forma de adaptar el modelo computacional y para ello se ha usado como base el modelo basado en prototipos [1][18], modelo que también emplean como base lenguajes dinámicos [19] como *Self* [3], *Moostrap* [9] y *Kevo* [15] y que se adapta mejor a las operaciones reflectivas, pudiendo considerarse a los lenguajes dinámicos como una evolución de los lenguajes basados en prototipos. El modelo basado en prototipos ofrece una solución al problema de versionado de esquemas mencionado anteriormente. Al prescindir del concepto de clase, siendo el objeto la abstracción principal, la modificación de la estructura de miembros de una instancia aislada se hace directamente, al ser el objeto el encargado de mantenerla y no existir un elemento externo que determine la misma. En este modelo, todo el comportamiento compartido por una serie de objetos (sus métodos) puede estar contenido en un objeto especial llamado *trait object*, mientras que la estructura común de los mismos (atributos) estaría contenida en otro objeto denominado *prototype object*. Por tanto, la modificación de un *trait object* afectaría a todos los objetos relacionados con el mismo, produciéndose una evolución de esquemas. En este modelo el equivalente a la instanciación en el modelo de clases se realiza precisamente clonando el mencionado *prototype object* y la relación de herencia es un mecanismo de delegación jerárquica de mensajes, en contraposición al mecanismo de concatenación usado por lenguajes estáticos [16], siendo pues una relación de asociación más dotada de una semántica adicional. Este modelo posee la misma expresividad que el modelo basado en clases [18], pudiéndose convertir de forma sencilla un conjunto de clases al otro modelo tal y como aparece en la figura 1. Es necesario pues mantener el modelo de clases de la máquina para permitir la compatibilidad con el código desarrollado, ampliándolo para que soporte las primitivas de reflexión de los lenguajes dinámicos. La adaptación llevada a cabo para ello emplea conceptos del modelo de prototipos para modificar el *SSCLI*, de manera que cuando se empleen capacidades reflectivas:

- Las clases serán *trait objects*, conteniendo el comportamiento compartido por todas las instancias y usando una evolución de esquema perezosa para

los atributos, que hace que no se incorporen a sus instancias hasta que son utilizados por ellas explícitamente.

- Los objetos serán tratados como prototipos, permitiéndose pues que agrupen tanto atributos como métodos propios a cada uno de ellos sin necesidad de modificar su clase.

4 MODIFICACIONES REALIZADAS

La máquina se ha modificado para soportar reflexión estructural, que ofrece un balance adecuado entre flexibilidad y coste de la implementación. De esta forma el sistema permite añadir, eliminar o modificar, a todo objeto y clase del sistema, cualquier atributo o método (creado o existente) en tiempo de ejecución. Además, estas operaciones reflectivas podrán ser empleadas explícitamente por cualquier lenguaje existente en la máquina, mediante llamadas a servicios que se ubican en la librería estándar, para no tener que modificar el propio lenguaje, o bien de forma transparente, mediante la modificación del código nativo que es generado por el sistema a partir del código intermedio al que se convierte cualquier programa de alto nivel, lo que permitiría la implementación de nuevos lenguajes que usasen directamente estas capacidades. Las modificaciones se han estructurado como sigue:

- `Información reflectiva`: La elevada rigidez de las estructuras internas de *Rotor*, derivada del alto grado de optimización interna de la máquina, imposibilita alterar físicamente la representación interna de un objeto, al tener limitaciones muy específicas en cuanto al espacio en memoria que pueden ocupar y desestabilizar este tipo de alteraciones el funcionamiento general de la misma. Por ello, toda la información reflectiva (nuevos miembros incorporados, modificaciones realizadas a miembros existentes, etc.) se ha incorporado ampliando un elemento que toda clase y objeto tiene asociado de forma privada y única para labores de sincronización (entre otras), llamado *SyncBlock*. De esta manera, cada elemento tendrá asociada su información reflectiva fácilmente accesible desde su propietario. El *SyncBlock* es ampliado con nueva funcionalidad, no perdiendo ninguna de las funciones que anteriormente poseía, al permanecer inalterado el código ya existente. La información añadida dinámicamente a los objetos se mantiene en estructuras de datos dinámicas optimizadas para su acceso eficiente en la memoria privada del motor de ejecución (Figura 2), usando los mecanismos que posee la propia máquina para que el recolector de basura no libere erróneamente aquella información que todavía esté en uso.
- `Operaciones reflectivas`: Para mantener la coherencia del modelo computacional modificado, la información reflectiva asociada a cada objeto o clase deberá consultarse siguiendo el mecanismo de herencia por delegación. Por ejemplo, cuando se le envíe un mensaje a un objeto o clase, se analizaría si éste posee un método que lo implemente, ejecutándolo si existe. En caso

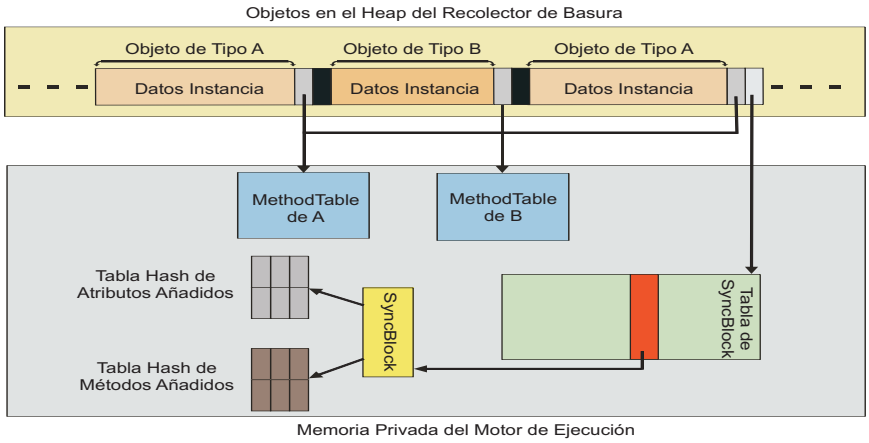

Fig. 2. Estructura de la nueva información reflectiva en *Rotor*

contrario, el proceso se repite en todos los objetos de la jerarquía de herencia del objeto o clase original. Dado que la información añadida dinámicamente en el *SyncBlock* puede sustituir a la existente en el objeto inicialmente, siempre es explorada primero cuando se intenta hacer cualquier acceso a la estructura del elemento. Esta disposición en memoria (Figura 2) resulta valida para ambos tipos de lenguajes. Si se procesa un lenguaje estático, que no hace uso de nuestras características de reflexión, se usa la información original del *SSCLI* (herencia por concatenación), mientras que, si se procesa un lenguaje dinámico, la información del *SyncBlock* se tratará siguiendo la estrategia ya mencionada. Por otra parte, para proporcionar un soporte adecuado de lenguajes dinámicos es necesario un sistema de tipos que pueda también ser dinámico, por lo que será necesario, a la hora de realizar el proceso de generación de código nativo, posponer ciertas comprobaciones de tipo para que se hagan en tiempo de ejecución, cambiando el sistema original. Un ejemplo de ello ocurre al compilar un código en el que se intente acceder a un miembro no existente (siempre de acuerdo con la información disponible en tiempo de compilación). En este caso se cambiará el mecanismo estándar de devolver un error de compilación por el de "crear" un miembro ficticio, que evite el error y permita seguir la compilación normalmente.

Este mecanismo esta diseñado para causar un impacto mínimo en el compilador, de manera que este módulo en sí no es modificado, sino mas bien "engañado" para permitir tratar los tipos tal y como se pretende. Posteriormente, cuando se disponga de toda la información de tipos en tiempo de ejecución, se harán las comprobaciones pertinentes equivalentes a las que se harían en la máquina original, empleando esa información suministrada dinámicamente para hacer el resto de las operaciones, en vez de la declarada estáticamente. La información reflectiva añadida a clases e instancias se estructura tal y como muestra en la figura 2.

- `Llamada a métodos`: Todo método destinado a ser añadido a una clase o

instancia estará creado previamente, lo que no limita las posibilidades de uso del sistema, ya que éste posee la capacidad de crear métodos nuevos mediante código. Añadirlo simplemente consistirá en guardar una referencia al mismo sobre el objeto al que se incorpora. A la hora de la llamada, como parte de las modificaciones al sistema de tipos de la máquina ya mencionadas, el sistema admite ahora internamente un cambio del objeto sobre el que estos métodos "añadidos" se ejecutan (*this*), de manera que el código del método trabaje efectivamente sobre el objeto que supuestamente lo tiene añadido, en vez de sobre el objeto sobre el que el método estaba declarado originalmente. De esta forma se logra la ilusión de añadir un método a una clase sin necesidad de hacer costosas copias de ninguna estructura de datos, haciendo que realmente un método no tenga necesariamente una clase "propietaria", de forma acorde al modelo basado en prototipos empleado.

- `Incorporación de la funcionalidad reflectiva al lenguaje intermedio o IL`: Los pasos anteriormente descritos conllevan el desarrollo de una serie de servicios sobre la información reflectiva, que finalmente permitirán implementar todas las primitivas necesarias, situados dentro del motor de ejecución. Se han modificado ciertas instrucciones pertinentes del *IL* (*ldfld*, *stfld*, *call*, *callvirt*,...) para que reutilicen estos servicios, de manera que todo programa pueda usar todas las operaciones reflectivas descritas de forma transparente. Para ello no se ha modificado el lenguaje en sí, sino el código nativo que el *JIT* genera para estas instrucciones y el propio método de generación del mismo. El código nativo modificado cambia accesos directos a las direcciones de memoria de los miembros manipulados por llamadas a nuestros servicios, que dan la oportunidad de interceptar y procesar adecuadamente todo acceso en tiempo de ejecución para poder conseguir nuestros fines, siempre conservando el mismo estado del sistema al final de la ejecución de cada instrucción. De esta forma, se consigue que las instrucciones del *IL* generen código nativo que pueda consultar toda la información existente en las estructuras de datos del sistema dinámicamente, usando los servicios vistos para tomar decisiones acerca del comportamiento del programa durante la ejecución en función de dicha información, y cambiar pues los valores devueltos por estas instrucciones para ajustarse a los datos disponibles en un momento concreto. Al usar los mismos servicios ya implementados, se consigue un tratamiento común de las operaciones reflectivas, independientemente del punto de acceso a las mismas (*IL* o librería estándar *BCL*). La figura 3 muestra la estructura de esta última modificación, donde puede apreciarse como, en tiempo de ejecución, el código nativo modificado que el sistema genera llama ahora a los nuevos servicios reflectivos añadidos al sistema, mediante la intercepción de los accesos directos a la memoria del programa para cambiarlos por llamadas a funciones a los mismos, procesando cada operación de acuerdo al modelo computacional visto.

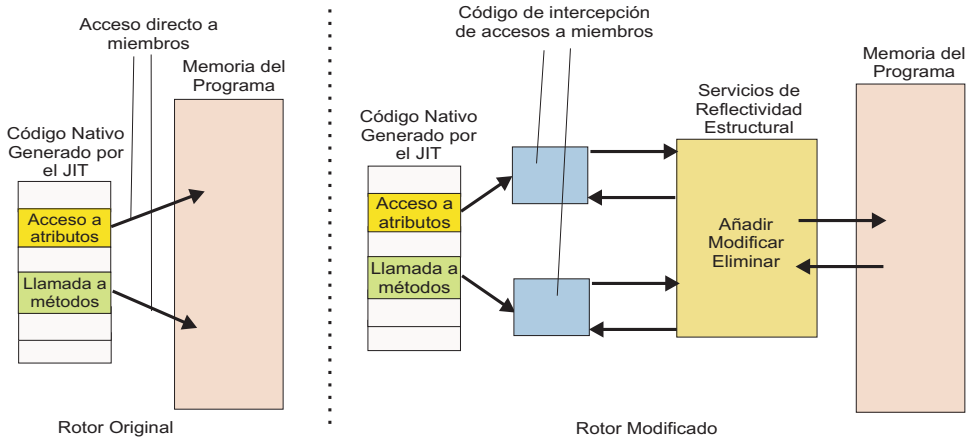

Fig. 3. Diagrama de las operaciones realizadas por el código nativo modificado

5 RESULTADOS

Para evaluar la validez de la idea presentada se emplearon tres estudios de rendimiento de nuestro sistema (denominado *RRotor*) frente a *Python*:

- Rendimiento de las primitivas de reflexión estructural implementadas, usando *benchmarks* que empleen operaciones reflectivas.
- El coste de la adaptabilidad, estudiando la penalización introducida por las modificaciones realizadas respecto al sistema original.
- Rendimiento en ejecución del código no reflectivo.

Nuestro sistema ha sido comparado con las últimas versiones de varias implementaciones de *Python* de diferente naturaleza. Por una parte, implementaciones como *CPython* y *ActivePython* son intérpretes desarrollados en *C*, mientras que distribuciones como *Jython* e *IronPython* generan código para la máquina virtual *Java* y *.NET* respectivamente, emulando sus características reflectivas sobre un lenguaje de alto nivel. Las primitivas reflectivas se han medido usando los diferentes escenarios que aparecen en la figura 4, usando 10.000 iteraciones en cada caso. En esta figura no se incluyen las dos últimas distribuciones por su rendimiento muy inferior (sus tiempos son del orden de varios segundos, no de milisegundos como el resto de sistemas).

Nuestra modificación de *SSCLI* es 3,17 veces más rápida que *ActivePython* y 3,14 veces más que *CPython*, aunque debe tenerse en cuenta que no se soporta el modelo computacional completo de este lenguaje. El único test en el que *RRotor* es ligeramente mas lento es el consistente en añadir métodos a clases. La diferencia se atribuye al modo en que se implementa la estructura para guardar la información de los mismos, que requiere la construcción de un nombre interno identificativo y el uso de un mecanismo para implementar las referencias al recolector de basura generacional de la máquina, que requiere el uso de indirecciones adicionales, lo que puede imponer un coste significativo. Las otras tres primitivas que ofrecen una diferencia de rendimiento

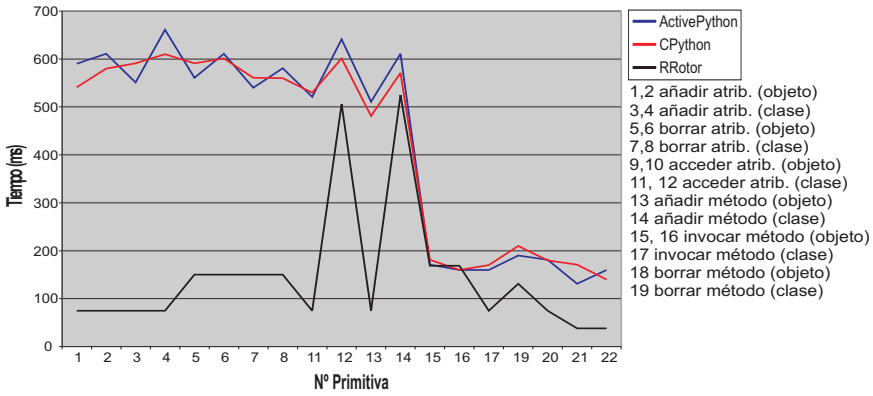

Fig. 4. Rendimiento de las primitivas reflectivas

menor frente a la implementación de *Python* de pruebas son las que acceden a miembros inexistentes. Esto es debido al coste impuesto por el algoritmo de búsqueda de miembros que permite implementar el modelo de herencia por delegación ya mencionado, cuya complejidad es elevada.

El coste de incorporar flexibilidad al sistema original se ha medido empleando el *benchmark Tommti* [2] sobre el *SSCLI* original y *RRotor*. Este *benchmark* no emplea ninguna de las operaciones reflectivas creadas, por lo que mide el rendimiento de ambos sistemas ante programas estáticos usando operaciones consistentes en la realización de diversos cálculos numéricos y sobre estructuras de datos comunes. En este caso, sólo se ha detectado una perdida significativa de rendimiento en los test que realizan accesos a miembros, no existiendo apenas pérdidas de rendimiento en otro tipo de tests. Tras realizar una serie de optimizaciones, se ha determinado que *RRotor* es alrededor 0,6 veces más lento en las operaciones que requieren el acceso a miembros (instrucciones *ldfld* y *call* respectivamente), debido a los cálculos adicionales necesarios para soportar el nuevo modelo computacional introducido. En este sentido se han empleado tambien dos *benchmarks* reales en *C#*, *LCSCBench* (*parser LR*, *front end* de un compilador de *C#*) y *AHCBench* (algoritmo de compresión adaptativa *Huffman*), resultando 0,81 y 2,14 veces más lentos en nuestro sistema respectivamente.

Por último, para comprobar el rendimiento de nuestro sistema frente al ofrecido por los lenguajes dinámicos, se ha ejecutado el *benchmark Tommti* de nuevo tanto en *RRotor* como en *CPython* (traduciendo convenientemente el código del *benchmark* a *Python*). El resultado obtenido es que nuestro sistema es 7,41 veces más rápido de media en la ejecución de código que no utiliza reflexión. Se ha encontrado una pérdida de rendimiento en operaciones relativas al acceso a miembros de clases que representan estructuras de datos comunes, operaciones de I/O y manejo de excepciones. En el primer caso, la pérdida de rendimiento se atribuye a que en *.NET* ésos elementos pertenecen a la *BCL*, mientras que en *Python* son parte del lenguaje, siendo más eficiente su acceso en el segundo caso. Para las excepciones, la perdida de rendimiento

se atribuye a un rendimiento pobre de este mecanismo en *SSCLI* y *CLR* [2].

6 TRABAJO FUTURO

Este proyecto puede servir como base para futuros aplicaciones relativas a la inserción dinámica de aspectos en tiempo de ejecución, el desarrollo rápido de prototipos o de *software* dirigido por modelos. Uno de los proyectos en funcionamiento actualmente es derivado de otro proyecto con *Microsoft Research*, a través de premio *RFP* de *Microsoft* concedido a una continuación de este proyecto, que empleará el sistema construido para, mediante el *framework Phoenix* [8] dotar al sistema de características dinámicas de alto nivel como metaclases, utilización de sistemas de tipos estáticos y dinámicos en un mismo lenguaje de programación [7] y primitivas que soporten programación orientada a aspectos dinámica.

7 CONCLUSIONES

En nuestro trabajo hemos establecido que la modificación de una máquina virtual que emplee compilación *JIT* para soportar de forma nativa primitivas de lenguajes dinámicos es una técnica que ofrece resultados satisfactorios de rendimiento, consiguiendo una eficiencia 7,4 veces mejor cuando se ejecuta código no reflectivo y 3,1 veces cuando el código es reflectivo, mostrándose superior a la ejecución de intérpretes y a la simulación de capacidades reflectivas con una capa de código adicional, técnicas usadas por otros sistemas que soportan estos lenguajes. Por otra parte, hemos visto como es posible integrar en la máquina un modelo computacional que soporte lenguajes "estáticos" y dinámicos de forma coherente, logrando pues un modelo que permita la potencial convivencia e interoperabilidad de ambos bajo la plataforma.

References

[1] Borning, A. H.. *Classes Versus Prototypes in Object-Oriented Languages*. Proceedings of the ACM/IEEE Fall Joint Computer Conference 36-40 (1986).

[2] Bruckschlegel, T.. "Microbenchmarking C++, C#, and Java". Dr. Dobbs Portal (2005). URL: http://www.ddj.com/dept/cpp/184401976.

[3] Chambers, C., D. Ungar, and E. Lee. *An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. OOPSLA 89 Conf. Proceedings. Published as SIGPLAN Notices, **24**, 10, 49-70 (1989).

[4] Ferg, S.. "Python & Java: a Side-by-Side Comparison". URL: http://www.ferg.org/projects/python_java_side-by-side.html.

[5] Golm, M., and J. Kleinder. *MetaJava - A Platform for Adaptable Operating-System Mechanisms*. Lecture Notes in Comp. Science 1357 (Springer-Verlag, 1997).

[6] González, S., W. De Meuter, P. Costanza, S. Ducasse, R. Gabriel, and T. DrsquoHondt.*2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era : Back to Dynamicity*. ECOOP 2004 Workshop Reader, Lect. Notes in C. Science. V. **3344** (2004).

[7] Meijer, E., and P. Drayton. *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages*. OOPSLA Workshop on Revival of Dynamic Languages (Proceedings) (2004).

[8] Microsoft Research. "Phoenix Framework". URL: http://research.microsoft.com/phoenix/.

[9] Mulet, P., and P. Cointe. "Definition of a Reflective Kernel for a Prototype-Based Language". International Symposium on Object Technologies for Advanced Software. Kanazawa (Japón) (1993).

[10] Object Mentor. "JUnit, Testing Resources for Extreme Programming". URL: http://www.junit.org.

[11] Ortin S., F. "Sistema Comp. De Prog. Flexible Diseñado Sobre Una Máquina Abstracta Reflectiva No Restrictiva". T. Doctoral. U. de Oviedo (2001).

[12] Roddick, J.. *A Survey of Schema Versioning Issues for Database Systems*. Information and Software Technology, **37** (1995).

[13] Stutz, D., T. Neward, and G. Shilling. "SSCLI Essentials". O'Reilly & Associates (2003).

[14] Sun Developer Network. "The Java HotSpot Virtual Machine". URL: http://java.sun.com/javase/technologies/hotspot.jsp.

[15] Taivalsaari, A.. *Kevo: A delegation-free prototype-based object-oriented language*. Proceedings of the 9th annual conference on Object-oriented programming systems, language, and applications (1994).

[16] Taivalsaari, A.. *Delegation versus concatenation or cloning is inheritance too*. ACM SIGPLAN Volume 6, Issue 3, pp. 20-49. July 1995.

[17] Tan, L., and T. Katayama. *Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution*. Proceedings of First International Conference on Deductive and OO Databases, DOOD, (1989).

[18] Ungar, D., G. Chambers, B. W. Chang, and U. Hlzl. "Org. Programs without Classes. Lisp and Symbolic Computation". Kluwer Academic Publishers (1991).

[19] Ungary D., and R. B. Smith. *SELF: The Power of Simplicity*. In OOPSLA 87 Conference Proceedings. SIGPLAN Notices, **22**, 12, 227-241 (1987).

[20] Van Rossum, G. "Python Language Reference Manual". Network Theory (2003).

[21] Zope Community. "What is Zope?". URL: http://www.zope.org/.